PowerBuilder Foundation Class Library User's Guide

PowerBuilder®

9

DOCUMENT ID: 37779-01-0900-01

LAST REVISED: March 2003

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

Contents

About This Book

Subject

This book describes how to use the PowerBuilder Foundation Class Library (PFC).

Audience

This book assumes that you:

- Are comfortable using Microsoft Windows applications
- Are currently developing applications with PowerBuilder and understand the concepts and techniques described in the *Applications Techniques* book
- Understand SQL and how to use your site-specific DBMS

This book has four parts, each for a specific group of PFC users:

Part	Title	Audience
1	PFC Overview	All PFC users
2	PFC Class Library Design	Object administrators
3	PFC Programming	Application developers
4	PFC Tutorial	All PFC users

Other sources of information

Use the Sybase Technical Library CD and the Technical Library Product Manuals web site to learn more about your product:

- The Technical Library CD contains product manuals and is included with your software. The DynaText reader (included on the Technical Library CD) allows you to access technical information about your product in an easy-to-use format.

  Refer to the *Technical Library Installation Guide* in your documentation package for instructions on installing and starting the Technical Library.

- The Technical Library Product Manuals web site is an HTML version of the Technical Library CD that you can access using a standard web browser. In addition to product manuals, you will find links to EBFs/Updates, Technical Documents, Case Management, Solved Cases, newsgroups, and the Sybase Developer Network.

  To access the Technical Library Product Manuals web site, go to Product Manuals at http://www.sybase.com/support/manuals/.

If you need help

Each Sybase installation that has purchased a support contract has one or more designated people who are authorized to contact Sybase Technical Support. If you cannot resolve a problem using the manuals or online help, please have the designated person contact Sybase Technical Support or the Sybase subsidiary in your area.

PART 1

PFC Overview

This part describes the PowerBuilder Foundation Class Library and prerequisite PowerBuilder concepts.

This part is for all PFC users.

CHAPTER 1 About the PowerBuilder Foundation Class Library

About this chapter

This chapter introduces the PowerBuilder Foundation Class Library (PFC). It includes PFC basics, prerequisite PowerBuilder concepts, object-oriented concepts, and a list of PFC components.

Contents

Understanding PFC

The PowerBuilder Foundation Class Library (PFC) is a set of PowerBuilder objects that you customize and use to develop class libraries. You can use these objects to provide corporate, departmental, or application consistency. PFC also includes objects that you use as is for utility purposes, such as debugging.

PowerBuilder objects

PFC is written in PowerBuilder and delivered as PowerBuilder objects with supporting PowerScript source code. It uses advanced PowerBuilder object-oriented coding techniques, and features a service-oriented design—that ensures that your application uses the minimum amount of computer resources.

Read the code
PFC uses many advanced PowerBuilder coding techniques. You can use the PowerBuilder PowerScript editor to examine the objects, instance variables, events, and functions in PFC ancestor objects.

What this book contains

This book explains PFC concepts (what things are and why you use them) as well as usage information (how to program using PFC).

For more information

For detailed information on PFC objects, instance variables, events, and functions, see the *PFC Object Reference*.

Understanding PowerBuilder

You use PFC to create advanced, object-oriented PowerBuilder class libraries. To get the most out of PFC and its object-oriented features, you must understand PowerBuilder and its object-oriented features. This section gives an overview of the PowerBuilder concepts with which you should be familiar.

Building PFC applications out of the box
PFC is designed primarily for building class libraries. But nothing prevents you from using PFC as is to build applications.

For complete information on PowerBuilder concepts, see the *PowerBuilder User's Guide*.

PowerBuilder libraries and objects

PFC is delivered as a set of PowerBuilder libraries (PBLs). These libraries contain the ancestor and descendent objects you use to write an application with PFC.

PowerBuilder libraries

Before you can use any PFC objects, you must add the PFC libraries to your application's library search path. PowerBuilder uses the library search path (which you define in the Target properties sheet) to find referenced objects during execution.

PowerBuilder objects

These are the main PowerBuilder objects you use with PFC:

PowerBuilder objects	Purpose
Windows	The interface between a user and a PowerBuilder application
Menus	Lists of commands that a user can select in the currently active window

PowerBuilder objects	Purpose
DataWindow objects	Used to retrieve, present, and manipulate data
User objects	Reusable components that you define once and use many times

There are two types of user objects:

- Visual user objects
- Class user objects

Visual user objects A visual user object is a reusable visual control or set of visual controls with a predefined behavior. PFC includes two types of visual user objects:

- **Standard visual user objects** PFC provides a full set of standard visual user objects. Each PFC standard visual user object corresponds to a PowerBuilder window control. These objects include predefined behaviors that provide complete integration with PFC services. In particular, the u_dw DataWindow user object, offers extensive functionality and integration with PFC services.
- **Custom visual user objects** PFC also use custom visual user objects. Custom visual user objects contain a group of window controls. These objects provide advanced functionality for use in specific situations.

PFC does not use external visual user objects. For complete information on visual user objects, see the *PowerBuilder User's Guide*.

Class user objects A class user object is a reusable nonvisual control you use to implement processing with no visual component. PFC includes two types of class user objects:

- **Standard class user objects** Inherit their definitions from built-in PowerBuilder system objects. PFC provides standard class user objects for transaction, error, and all other extendable system objects.
- **Custom class user objects** Inherit their definitions from the PowerBuilder NonVisualObject class. Custom class user objects encapsulate data and code. This type of class user object allows you to define an object class from scratch.

  PFC uses custom class user objects to implement many of its services and provides functions to enable instances of these service objects.

  It also provides **reference variables**, which are pointers to an instantiated object. You use a reference variable to access an object's instance variables, functions, and events.

Functions

PowerBuilder supports global functions and object functions. PFC performs much of its processing through user-object functions. A **function** is a collection of PowerScript statements that perform some processing. You pass zero or more arguments to a function, and it may return a value.

For complete information on PFC object functions, see the *PFC Object Reference*.

Events and user events

Windows, user objects, and controls each have a predefined set of events. PFC extends this by defining user events for many PFC objects. Events can accept arguments and may return a value.

There are three types of PFC events:

This type of event	Executes when the user performs
Predefined PowerBuilder events	An action that causes the operating system to invoke the event
Predefined user event	An action (such as selecting a menu item) that causes PFC to trigger the user event
Empty user events (you add PowerScript code)	An action (such as selecting a menu item) that causes PFC to trigger the user event

You can also add code to call, trigger, or post predefined events and user events.

In this book
Unless otherwise qualified, this book uses the word *event* to refer to all three types.

Functions and events compared

Functions and events are similar in many ways: they may accept arguments and return values; they both consist of PowerScript statements; they can be called, triggered, and posted. But there are some differences between functions and events:

Feature	Functions	Events
Call to nonexistent method	Invoking a nonexistent function at runtime produces an error	Invoking a nonexistent event with TriggerEvent yields a return value of -1
Processing of ancestor script	Functions override ancestor processing (although they can call ancestor functions using the Super keyword)	Events can extend or override ancestor processing

Feature	Functions	Events
Access	Object functions can be public, private, or protected	Events always have public access
Overloading	Functions of the same name can take different arguments	Events cannot be overloaded

Object-oriented programming

Object-oriented programming tools support three fundamental principles: inheritance, encapsulation, and polymorphism.

Inheritance

Inheritance means that objects can be derived from existing objects, with access to their visual component, data, and code. Inheritance saves coding time, maximizes code reuse, and enhances consistency.

Encapsulation

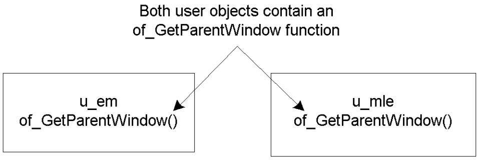
Encapsulation (also called information hiding) means that an object contains its own data and code, allowing outside access as appropriate. PFC implements encapsulation as follows:

- PFC defines object functions and instance variables as public or protected, depending on the desired degree of outside access. PFC does not use the private access level.
- For readable instance variables, PFC generally provides an *of_Getvariablename* function.
- For Boolean instance variables, PFC generally provides an *of_Isvariablename* function.
- For modifiable instance variables, PFC generally provides an *of_Setvariablename* function.
- In certain cases, PFC defines an instance variable as public, allowing you to access it directly.

Polymorphism

Polymorphism means that functions with the same name behave differently depending on the referenced object and the number of arguments. PFC supports the following types of polymorphism:

- With **operational polymorphism**, separate unrelated objects define a function with the same name:

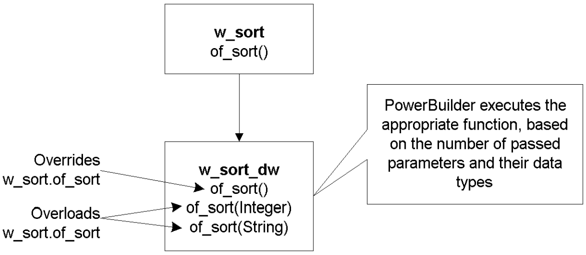


- With **inclusional polymorphism**, various objects in an inheritance chain define a function with the same name but different arguments:

Overriding and overloading

PowerBuilder supports both **function overriding** or **function overloading**:

- In function *overriding*, the descendent function has the same arguments or argument data types.
- In function *overloading*, the descendent function (or an identically named function in the same object) has different arguments or argument data types.



How PFC uses object orientation

PFC uses all facets of PowerBuilder’s object-oriented capabilities.

Principles

PFC uses the three principles of object orientation:

PFC uses	To
Inheritance	Implement a hierarchy of windows, menus, and user objects
Encapsulation	Isolate each object’s data and code
Polymorphism	Provide same-named functions (within one object, within an inheritance hierarchy, and among multiple objects)

Services

PFC uses windows, standard class user objects, and custom class user objects to implement an object-oriented design by isolating related types of processing (such as DataWindow caching, row selection, and window resizing). These related groups of processing are called **services**. Most services are implemented as custom class user objects. PFC service types include:

Service category	Service
Application services	Application preferences
	DataWindow caching
	Debug
	Error message
	Most recently used object
	Security
	Transaction registration
Window services	Base
	Preferences
	Sheet manager
	Status bar
DataWindow services	Base
	DataWindow resize
	Drop-down search
	Filter
	Find
	Linkage
	Multitable update
	Properties
	Querymode
	Report
	Required column
	Resize
	Row manager
	Row selection
	Sort
DataStore services	Base
	Multitable update
	Print preview
	Report

Service category	Service
Global services	File
	INI file
	Logical unit of work
	MetaClass
	Menu
	Numerical
	Platform
	Resize
	RTE find
	Selection
	SQL
	SQL Spy
	String

Enabling services selectively Selectively instantiating service objects provides you with complete flexibility in the PFC functionality used by your application—and allows your applications to use fewer resources. PFC automatically destroys all service objects created by an application.

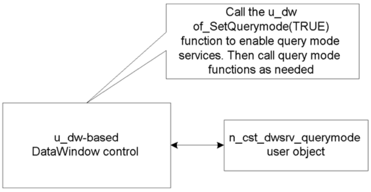
Enabling services selectively has many benefits, including:

- Minimizing the number of ancestor objects typically found in a deep inheritance chain
- Minimizing application overhead (use only the services you need)
- Building both simple and complex applications
- Ease of use and maintenance (you do not have to write multiple scripts to override ancestor processing)

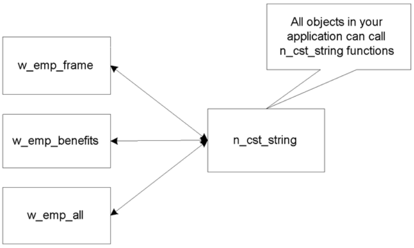
Delegation PFC's service orientation reflects the object-oriented concept of **delegation**, which divides the main object and its implementation into separate object hierarchies.

PFC uses two types of relationships for delegation:

- **Aggregate relationship** The service object cannot function apart from its owning object. This is sometimes called a whole-part relationship. For example, the u_dw DataWindow visual user object uses the n_cst_dwsrv_querymode user object for query mode services:


Call the u_dw of_SetQuerymode(TRUE) function to enable query mode services. Then call query mode functions as needed
u_dw-based DataWindow control
n_cst_dwsrv_querymode user object

- **Associative relationship** The service object can function alone. For example, string services are provided by the n_cst_string user object and are available to objects throughout your application:


All objects in your application can call n_cst_string functions
w_emp_frame
w_emp_benefits
w_emp_all
n_cst_string

For more information on PFC service types and how to use them, see Chapter 4, “Using PFC Services”.

How PFC uses the extension level

No class library can meet your needs right out of the box. You typically modify PFC objects to integrate application-wide functions and objects. Without the PFC extension level, this could present a problem whenever a new version of PFC is released: applying the new version would overwrite your customizations, forcing you to reapply these changes manually.

A separate extension level

PFC implements an **extension level** in all its inheritance hierarchies. All extension objects reside in separate PBLs, which are not affected when you upgrade to the latest version:

Contents	Ancestor level	Extension level
Application and global services	PFCAPSRV.PBL	PFEAPSRV.PBL
DataWindow services	PFCDWSRV.PBL	PFEDWSRV.PBL
Visual and standard class user objects	PFCMAIN.PBL	PFEMAIN.PBL
Utility services	PFCUTIL.PBL	PFEUTIL.PBL
Window services	PFCWNSRV.PBL	PFEWNSRV.PBL

Objects in the ancestor-level libraries contain all instance variables, events, and functions; objects in the extension-level libraries are unmodified descendants of corresponding objects in the ancestor library. But through inheritance they have access to the ancestor's instance variables, events, and functions.

To see the instance variables, events, and functions available to a descendent object, use the PowerBuilder Browser.

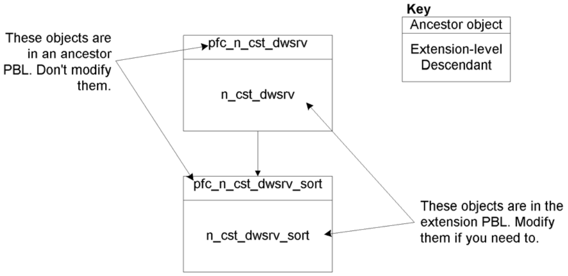
Using an extension level has two major advantages:

- You can add site-, department-, and application-specific logic to extension level objects
- The extension PBLs are not affected when upgrading to the latest version

Obsolete objects
The PFCOLD.PBL library contains obsolete objects. If you have an existing PFC application, you may need to add this library to your application target library list.

What you do

You customize your PFC application by modifying objects at the extension level. *You do not modify ancestor objects.* Your application's objects use extension-level user objects and inherit from extension-level windows:


Key
Ancestor object
Extension-level Descendant
These objects are in an ancestor PBL. Don't modify them.
pfc_n_cst_dwsrv
n_cst_dwsrv
pfc_n_cst_dwsrv_sort
n_cst_dwsrv_sort
These objects are in the extension PBL. Modify them if you need to.

The PFC object-naming convention

PFC uses the following object-naming convention:

Level	Name	Contains
Ancestor objects	Use the prefix pfc_	All instance variables, events, and functions
Extension-level objects	Have the same name as their ancestor but without the prefix pfc_	Unmodified descendants of PFC ancestor objects

For example, the ancestor for the DataWindow selection service object is pfc_n_cst_dwsrv; the extension-level descendant is n_cst_dwsrv. Pfc_n_cst_dwsrv contains all code for the service; n_cst_dwsrv is an unmodified descendant to which you may add application-specific instance variables and code.

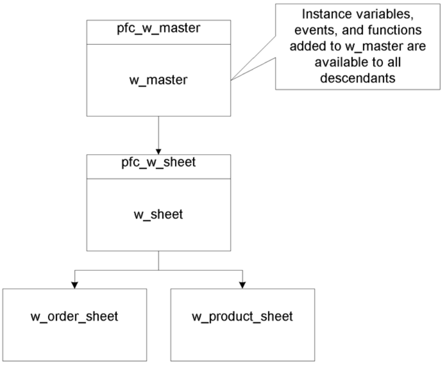
PFC-defined user events PFC-defined user events also use the pfc_ prefix. This makes it easy for you to distinguish your application's user events from PFC's user events.

PFC documentation uses extension-level names
PFC documentation always uses the extension-level name when referring to a service object. For example, this book refers to w_master when discussing the base-class window, not to pfc_w_master. But it's important to remember that the instance variables, events, and functions available to w_master are actually defined in pfc_w_master.

For complete information on PFC object-naming conventions, see the *PFC Object Reference*.

Sample extension scenario

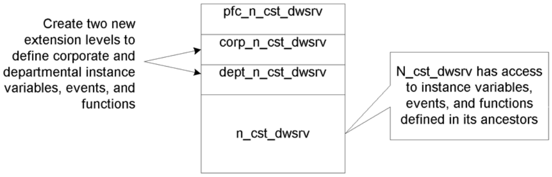
PFC's object hierarchies allow you to add extension logic at each level. Because pfc_w_sheet inherits from w_master, for example, instance variables, functions, and events you add to w_master are available to all descendent windows:


pfc_w_master
w_master
Instance variables, events, and functions added to w_master are available to all descendants
pfc_w_sheet
w_sheet
w_order_sheet
w_product_sheet

Adding extension levels

The extension layer provides for reusability within an application and effectively insulates individual applications from PFC upgrades. But large installations that have department-wide (and perhaps corporate-wide) standards must extend this strategy further to implement additional levels containing corporate and departmental standards and business rules.

If you are using PFC in an organization, you may want to create additional extension levels to contain corporate or departmental variables, events, and functions. Applications still use objects in PFC extension libraries but now have access to additional ancestor instance variables, events, and functions:


Create two new extension levels to define corporate and departmental instance variables, events, and functions
pfc_n_cst_dwsrv
corp_n_cst_dwsrv
dept_n_cst_dwsrv
n_cst_dwsrv
N_cst_dwsrv has access to instance variables, events, and functions defined in its ancestors

The PFC components

PFC is made up of the following:

- A set of PBLs (libraries)

  You must ensure that the objects in these PBLs are available to PFC-based applications by adding them to the application target library list.

- A database
- Code examples
- A sample application

The PFC PBLs

PFC is distributed with PBLs containing ancestor objects and PBLs containing extension-level objects. Each ancestor level/extension level set contains objects that perform related services:

Libraries	Contents
PFCAPSRV.PBL PFEAPSRV.PBL	Application manager, application service objects, and other global service objects
PFCDWSRV.PBL PFEDWSRV.PBL	DataWindow services, including user objects and utility windows
PFCMAIN.PBL PFEMAIN.PBL	Standard visual user object, custom visual user object, and standard class user objects
PFCUTIL.PBL PFEUTIL.PBL	Utility objects and services
PFCWNSRV.PBL PFEWNSRV.PBL	Window services, including user objects, and utility windows
PFCOLD.PBL	Obsolete PFC objects (base and extension-level objects)

Use the Library painter
Use the PowerBuilder Library painter to see a list of all objects in PFC libraries.

The PFC database

PFC ships with the pfc.db local database. This database contains the following tables:

Table	Usage
Messages	Error message service
Security_apps	Security service
Security_groupings	Security service

Table	Usage
Security_info	Security service
Security_template	Security service
Security_users	Security service

The PFC local database is intended for developer use only. If your application uses the error message service or security service, you should copy these tables to a server database, as described in “Deploying database tables” on page 229.

The PFC code examples

Use the PFC code examples to view PFC objects and services in action and learn how to code and implement most common PFC functionality. The PFC code example interface provides extensive cross-reference and usage information.

The PFC sample application

Use PEAT (the PFC sample application) to see an example of PFC used in a project estimation and tracking system.

PART 2

PFC Class Library Design

This part describes how to extend PFC to create your own class library.

This part is for object administrators—those responsible for class library maintenance, enhancement, and implementation.

CHAPTER 2

Designing a Class Library

About this chapter

This chapter explains how to use PFC as the basis for your own class library.

Contents

Using PFC to design a class library

PFC is a foundation upon which you build class libraries, leveraging PFC's extensible service-oriented architecture to customize behavior and extend capabilities—and even define your own services.

The object administrator

PowerBuilder users who use PFC to design class libraries are **object administrators**. Object administrators can be:

- **Corporate and departmental analysts** Create PFC-based class libraries to enable consistency and enhance functionality
- **Consultants** Create PFC-based class libraries to add value to their services
- **Vendors** Use PFC as the basis for advanced class libraries that meet the needs of a specific set of developers

Your role

As object administrator, you will use PFC in a different way from developers. You will customize and enhance PFC functionality for use by all developers. You need a thorough understanding of PFC, as well as of your organization's needs. Based on anticipated usage, you will extend PFC by adding and customizing objects, services, instance variables, events, and functions.

Your first step is to choose a PFC extension strategy.

Choosing an extension strategy

Although there are many ways in which sites extend and implement PFC, there are two main PFC extension strategies:

- Create an intermediate extension level
- Use the existing PFC extension level

Using separate physical files

Regardless of strategy, each PFC application should have its own set of physical files. You cannot share ancestor files (those whose name starts with PFC). This is because of internal interdependencies of high-level extension objects, such as w_master.

For example, assume that applications 1 and 2 have their own sets of PFE extension-level libraries but share ancestor libraries. Application 1 adds a function of_SetData to w_master in its version of PFEMAIN.PBL; this function is available to all descendants of w_master, including pfc_w_main, pfc_w_frame, and pfc_w_sheet in the shared ancestor libraries. Application 2 then regenerates the application.

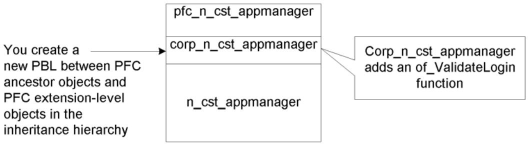
Because application 2 has no of_SetData function in its PFEMAIN.PBL, all internal references to of_SetData are removed from w_master descendants, resulting in execution time and compiler errors for application 1.

Creating an intermediate extension level

To create objects that accommodate corporate or departmental usage but that also allow developers to freely add application-wide code to the extension level, you can define one or more intermediate extension levels. These intermediate extension-level objects contain site- and department-specific instance variables, events, and functions.

Following this strategy, you create a new extension level between the PFC ancestor level and the PFC extension level. Then you redefine the PFC extension hierarchy so that intermediate extension level objects descend from PFC ancestor objects and PFC extension level objects descend from objects in the intermediate extension level. Because PFC objects use the data type of PFC extension level objects when declaring reference variables, these changes become available immediately.

For example, you might create a customized descendant of pfc_ n_cst_appmanager:


pfc_n_cst_appmanager
corp_n_cst_appmanager
n_cst_appmanager
You create a new PBL between PFC ancestor objects and PFC extension-level objects in the inheritance hierarchy
Corp_n_cst_appmanager adds an of_ValidateLogin function

Advantages

This strategy has two advantages:

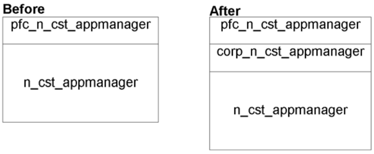
- Changes made to the objects in the intermediate extension level are available to descendent objects in the PFC extension level
- The developer has complete control over the PFC extension level

Naming standards
You should give objects in the intermediate extension level a standard prefix that reflects their usage. For example, if the intermediate extension level contains additional class library functionality, use classlib_ as the prefix; if the intermediate extension level contains corporate extensions, use corp_ as the prefix.

What you do

You add, modify, and extend PFC through objects in the intermediate extension level.

To implement an intermediate extension level, you can use the Library Extender or create intermediate extension level objects manually.



Use the Library Extender
It’s best to use the Library Extender to create intermediate extension levels.

See “Library Extender” on page 225.

v **To manually create an additional extension level and redefine the inheritance hierarchy:**

1 Create a PBL to contain intermediate extension level objects.

2 Define objects in the intermediate extension level (by inheriting from objects in the PFC ancestor level) and define instance variables, functions, and events as necessary. You can also define new objects in the intermediate extension level.

3 Define instance variables, events, and functions in the extension level objects as necessary.

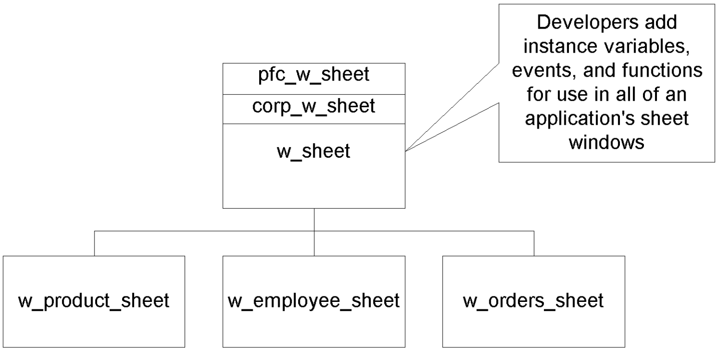
4 Redefine the inheritance hierarchy by creating new PFC extension-level objects that inherit from the newly defined extension level (instead of inheriting from the PFC ancestor level).

How developers work

This type of extension level usage gives developers complete control of the PFC extension level. They can:

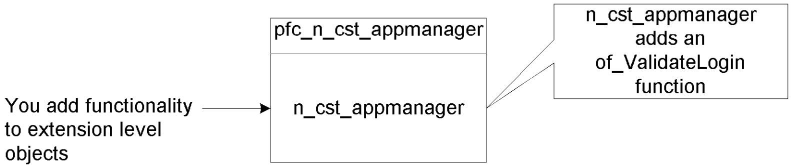
- Modify and use extension-level user objects
- Modify and inherit from extension-level windows, including w_master
- Modify and inherit from extension-level menus (optionally using them directly)

For example, developers might add application-specific functionality to w_sheet and use it as the ancestor for all sheet windows:


pfc_w_sheet
corp_w_sheet
w_sheet
Developers add instance variables, events, and functions for use in all of an application's sheet windows
w_product_sheet
w_employee_sheet
w_orders_sheet

Using the existing PFC extension level

Following this strategy, you add corporate or departmental modifications to the PFC extension level. *Developers do not use the extension level.*


pfc_n_cst_appmanager
n_cst_appmanager
You add functionality to extension level objects
n_cst_appmanager adds an of_ValidateLogin function

Advantages

This strategy has two advantages:

- Developers can share a common set of PBLs during the development phase
- Users can share a common set of PBDs when deploying multiple PFC applications

Disadvantages

This strategy has several disadvantages:

- Developers cannot modify extension level objects. Changes are lost when the object administrator creates a new version
- There is only one extension level. Corporate, departmental, and class library extensions must all exist at the same level
- Developers cannot extend extension level objects. Because PFC uses the data type of PFC extension level objects when declaring reference variables, changes to descendent objects are not automatically available throughout PFC

What you do

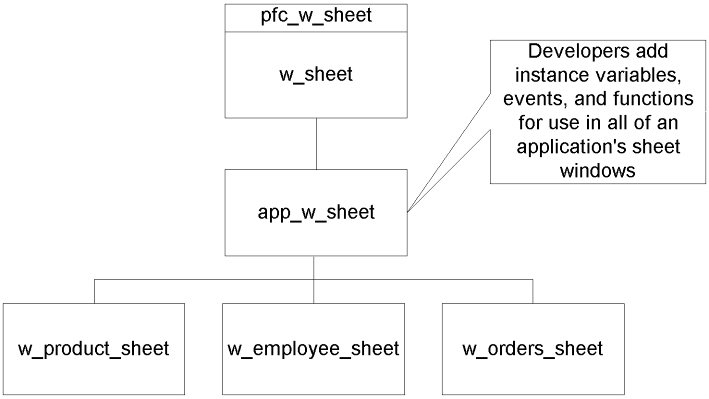
You add, modify, and extend PFC through objects in the PFC extension level.

To create a new service, you add the object to the extension level and there is no need for an ancestor object.

How developers work

When working with this extension strategy, developers use PFC services but do not modify any objects in the PFC extension level.

All developer extensions must be done through inheritance. For example, developers might add application-specific functionality to an application sheet window and use it as the ancestor for all sheet windows:


pfc_w_sheet
w_sheet
app_w_sheet
Developers add instance variables, events, and functions for use in all of an application's sheet windows
w_product_sheet
w_employee_sheet
w_orders_sheet

Defining a new service

After researching your requirements, you may need to define a new service. This might be an associative service (working with a main object such as a DataWindow, DataStore, or window) or an aggregate service for use anywhere in an application.

Where to define it
If you are *using the existing PFC extension level*, define your new service in one or more separate PBLs.

If you are *creating an extension level*, define your ancestor object in the intermediate extension level (the corporate level in the example ahead) and your extension level object in the PFC extension level.

v **To define an associative service:**

1 Create a custom class user object that contains the necessary instance variables, functions, and events. For example, define a DataWindow service to perform automatic row insertion. In this case, the object should inherit from n_cst_dwsrv.

2 Save this user object in the intermediate extension level. For example, if you have an intermediate level for corporate objects, it might be named corp_n_cst_dwsrv_autorowinsert.

3 Create a descendent user object in the extension level. For example, n_cst_dwsrv_autorowinsert. Add no code to this object.

4 Add an instance variable to the main object in the intermediate extension level. This variable should use the data type of the extension level object. For example, update corp_u_dw by adding an inv_autorowinsert instance variable of type n_cst_dwsrv_autorowinsert.

5 Add a function to the main object in the intermediate extension level. This function should create or destroy an instance of your user object as specified by a passed boolean argument. For example, of_SetAutoRowInsert might create or destroy an instance of n_cst_dwsrv_autorowinsert as follows:

```
// Function name: of_SetAutoRowInsert
// Arguments: ab_switch (boolean by value)
// Returns: 1 = Success
//          0 = Already instantiated
//         -1 = Argument was NULL
IF IsNull(ab_switch) THEN
  Return -1
END IF
IF ab_switch THEN
  IF IsNull(inv_autorowinsert) OR  &
    NOT IsValid(inv_autorowinsert) THEN
     inv_autorowinsert = CREATE n_cst_autorowinsert
     // of_SetRequestor = defined in ancestor
     inv_autorowinsert.of_SetRequestor(this)
     Return 1
  END IF
ELSE
  IF IsValid(inv_autorowinsert) THEN
    DESTROY inv_autorowinsert
    Return 1
  END IF
END IF
Return 0
```

6 Add code to the Destructor event of the main object in the intermediate extension level. This code should destroy your user object:

```
this.of_SetAutoRowInsert(FALSE)
```

7. Add code as necessary to events in the main object. This code should call events on the service object, if enabled. For example, the automatic row insertion service might add the following code to the RowFocusChanging event:

```
IF IsValid(inv_autorowinsert) THEN
   inv_autorowinsert.Event corp_FocusChanging &
      (currentrow, newrow)
END IF
```

v **To define an aggregate service:**

1. Create a custom class user object that contains the necessary instance variables, functions, and events. Optionally assign the AutoInstantiate property to this object.
2. Save this user object in the intermediate extension level.
3. Create a descendent user object in the extension level. Add no code to this object.
4. Use this object in event and function scripts as necessary. Reference variables for this object should use the data type of the extension level object.

PART 3

PFC Programming

This part explains how to program using PFC and PFC services.

This part is for application developers.

CHAPTER 3 PFC Programming Basics

About this chapter

This chapter explains basic PFC programming practices and tells you how to get started with a PFC application.

Assumptions
This chapter and all remaining chapters in this manual assume an intermediate extension level strategy, which allows the developer to modify and extend objects in the PFC extension level.

Contents

Setting up the application manager

The first step in creating an application with PFC is configuring and enabling the application manager, n_cst_appmanager. Within the application manager, you code logic that would otherwise be in the Application object.

The application manager also has instance variables and functions to maintain application attributes, such as the frame window, application and user INI files or registry keys, and the application Help file.

v **To set up the application manager:**

1. Define an application target library list that contains PFC PBLs:

   PFCAPSRV.PBL
   PFCDWSRV.PBL
   PFCMAIN.PBL
   PFCUTIL.PBL
   PFCWNSRV.PBL
   PFEAPSRV.PBL
   PFEDWSRV.PBL
   PFEMAIN.PBL
   PFEUTIL.PBL
   PFEWNSRV.PBL

   PFCOLD.PBL
   If your application uses obsolete PFC objects from a previous release of PFC, include PFCOLD.PBL in the library list.

2. From the Application painter, display the Variable view and declare a global variable, gnv_app, of type n_cst_appmanager:

   ```
   n_cst_appmanager   gnv_app
   ```

   The variable name must be gnv_app
   PFC objects, functions, and events require that you define the application manager as gnv_app, with a data type of n_cst_appmanager (or an n_cst_appmanager descendant).

3. In the painter Script view, add PowerScript code to your application's Open event to create n_cst_appmanager and call the pfc_Open event:

   ```
   gnv_app = CREATE n_cst_appmanager
   gnv_app.Event pfc_Open(commandline)
   ```

4. Add code to your application's Close event to call the pfc_Close event and destroy n_cst_appmanager:

   ```
   gnv_app.Event pfc_Close( )
   DESTROY gnv_app
   ```

5. Add code to your application's SystemError event to call the pfc_SystemError event:

   ```
   gnv_app.Event pfc_SystemError( )
   ```

6. Close the Application painter and save the changes.

7 Display the User Object painter and open n_cst_appmanager, found in PFEAPSRV.PBL. (Optionally, use an application-specific descendant of n_cst_appmanager.)

8 Call n_cst_appmanager functions in the Constructor event to initialize instance variables for version, company, and INI file.

9 Call n_cst_appmanager functions in the pfc_Open event to enable the application services you want:

To enable this service	Call this function
Application preference	of_SetAppPreference
DataWindow caching	of_SetDWCache
Error	of_SetError
Most recently used object	of_SetMRU
Transaction registration	of_SetTrRegistration
Security	of_SetSecurity
Debug	of_SetDebug

10 Add code to the pfc_Open user event to open your application's initial window (typically the frame window), optionally including a call to the of_Splash function, which displays a splash screen.

11 (Optional) Add code to the pfc_PreAbout, pfc_PreLogonDlg, and pfc_PreSplash events to customize elements of the About box, logon dialog box, and splash screen.

12 (Optional) Add code to the pfc_Idle, pfc_ConnectionBegin, and pfc_ConnectionEnd events. If so:

- Call pfc_Idle from the application's Idle event.
- Call pfc_ConnectionBegin from the application's ConnectionBegin event.
- Call pfc_ConnectionEnd from the application's ConnectionEnd event.

13 Save n_cst_appmanager.

v **To display a splash screen:**

- Call the of_Splash function just before opening the initial window in the pfc_Open event:

```
this.of_Splash(1)
Open(w_tut_frame)
```

v **To display a logon screen:**

1 Call the of_LogonDlg in the frame window’s Open event:

```
Integer  li_return
li_return = gnv_app.of_LogonDlg( )
IF li_return = 1 THEN
      this.SetMicroHelp("Logon successful")
ELSE
      MessageBox("Logon", "Logon failed")
      Close(this)
END IF
```

Of_LogonDlg displays the w_logon dialog box, which prompts for user ID and password and calls the n_cst_appmanager pfc_Logon event when the user clicks OK.

Alternatively, you can call the of_LogonDlg function in the n_cst_appmanager pfc_Open event, immediately after opening the frame window. Do not call of_LogonDlg immediately after calling of_Splash.

2 Add script to the n_cst_appmanager pfc_Logon event to log the user on to the database. This example assumes an INI file that contains all information except user ID and password; it also assumes you’ve associated SQLCA with n_tr, PFC’s customized Transaction object:

```
Integer  li_return
String  ls_inifile, ls_userid, ls_password

ls_inifile = gnv_app.of_GetAppIniFile()
IF SQLCA.of_Init(ls_inifile,"Database") = -1 THEN
      Return -1
END IF
// as_userid and as_password are arguments
// to the pfc_Logon event
SQLCA.of_SetUser(as_userid, as_password)
IF SQLCA.of_Connect() = -1 THEN
      Return -1
ELSE
      gnv_app.of_SetUserID(as_userid)
      Return 1
END IF
```

Building applications

Building MDI applications

To build an MDI application with PFC, use the w_frame and w_sheet windows as the ancestors for your frame and sheet windows. To define events, functions, and instance variables for all your application's sheets, add them to w_sheet.

You must also define menu items for all sheet windows in an ancestor sheet menu (m_master, m_frame, or an application-specific sheet-menu ancestor, depending on your menu strategy).

For information on the strategies you can use to implement menus under PFC, see "Using menus with PFC" on page 201.

v **To build an MDI application with PFC:**

1 Add application-specific modifications to w_frame, optionally creating a frame window that inherits from w_frame.

2 (Optional) Add ancestor instance variables, functions, and user events to w_sheet.

3 Create sheet windows that inherit from w_sheet.

4 Create a frame menu according to your menu strategy, optionally using m_frame, PFC's frame menu.

5 Associate the frame window with the customized frame menu.

6 Create sheet menus according to your menu strategy.

7 Associate sheet windows with sheet menus.

8 Open the frame window in the n_cst_appmanager pfc_Open user event.

9 (Optional) Enable frame window services as necessary:

- Enable the status bar service by calling the w_frame of_SetStatusBar function
- Enable the sheet manager service by calling the w_frame of_SetSheetManager function

v **To open sheet windows in an MDI application:**

1 Add code to the Clicked event for the menu items that open sheet windows. This code should assign the sheet window name to Message.StringParm and call the of_SendMessage function, passing the pfc_Open event name:

```
n_cst_menu lnv_menu
```

```
Message.StringParm = "w_products"
lnv_menu.of_SendMessage(this, "pfc_Open")
```

2 Add code to the w_frame pfc_Open event that accesses Message.StringParm and opens the specified sheet window:

```
String  ls_sheet
w_sheet lw_sheet

ls_sheet = Message.StringParm
OpenSheet(lw_sheet, ls_sheet, this, 0, Layered!)
```

Building SDI applications

To build an SDI application with PFC, use the w_main window as the ancestor for your main windows. To implement events, functions, and instance variables so they are available in all windows, add them to w_main.

If your windows use menus, you must also define menus for each window.

For information on the strategies you can use to implement menus under PFC, see “Using menus with PFC” on page 201.

v **To build an SDI application with PFC:**

1 Create a main window that inherits from w_main, optionally modifying w_main directly.

2 Create a main menu according to your menu strategy.

3 Create additional windows and menus as appropriate.

4 Open the main window in the n_cst_appmanager pfc_Open user event.

Programming using PFC functions

Almost all PFC functions are object functions. This means they are defined within a PowerBuilder object (Window, Menu, or user object). Encapsulating functions within a PowerBuilder object enables you to quickly see which functions apply to the object.

PFC uses the Set/Get/Is naming convention to control access to instance variables:

- of_Set functions allow you to set the value of an instance variable
- of_Get functions allow you to access a nonboolean instance variable
- of_Is functions allow you to determine the state of a boolean instance variable

Other types of instance variable access
PFC also declares certain instance variables as public, allowing you to access them directly. Additionally, some variables are for internal use only and are not accessible via function call.

In addition to the Set/Get/Is convention, PFC uses a Register/UnRegister convention when defining a set of entities to be affected by a service. For example, you call the u_calculator object's of_Register function to define the DataWindow columns that use a drop-down calculator.

Object qualification PFC uses access levels (public, private, protected) to control your access to functions designed for internal use.

When you call these functions from outside the object, use dot notation to qualify the function name. Qualify the function name with the reference variable used to create the object (in some cases you qualify the function name with the actual object name).

v **To call PFC object functions:**

1 Ensure that the object has been created.

PowerBuilder creates windows, menus, and visual user objects when the window opens. You create most class user objects using an of_Set*servicename* function (defined in u_dw, n_cst_appmanager, w_master, or u_dw).

For example, the following u_dw object function creates the sort service (n_cst_dwsrv_sort user object) and saves a reference to it in u_dw's inv_sort instance variable. You typically code these functions in the DataWindow's Constructor event:

```
this.of_SetSort(TRUE)
```

Autoinstantiated objects
Certain PFC objects use PowerBuilder's autoinstantiate feature. These objects have no Set functions; PowerBuilder instantiates them automatically when you declare them as variables.

2 Call object functions from your application, as appropriate.

This example specifies that the sort service will use DataWindow column header names, sort on displayed values, implement point-and-click sorting, and display a drag-drop style dialog box when the user selects View>Sort from the menu bar:

```
this.inv_sort.of_SetColumnNameSource &
       (this.inv_sort.HEADER)
this.inv_sort.of_SetUseDisplay(TRUE)
this.inv_sort.of_SetColumnHeader(TRUE)
this.inv_sort.of_SetStyle &
       (this.inv_sort.DRAGDROP)
```

Function overloading PFC uses function overloading to provide a rich, flexible application programming interface. It implements function overloading in two ways:

- **Multiple syntaxes** Multiple functions contain arguments that use different data types or are in a different order. This allows PFC to handle many types of data in function arguments
- **Optional arguments** Multiple functions contain an increasing number of arguments with the same data types and in the same order. This allows PFC to provide defaults for commonly used arguments

Overloaded functions for internal use only
In addition to a series of Public overloaded functions, PFC often provides a Protected version, which the other versions call internally. For example, the n_cst_dwsrv_report of_AddLine function has four Public versions, and one Protected version that is called by the other four. Although you can call Protected versions in some cases, they are intended for internal use only and are subject to change.

Programming using PFC events

PFC includes precoded events and user events, which perform processing to implement PFC services. It also includes empty user events, which allow you to add application-specific code to perform application-specific tasks.

All events have public access and you can use dot notation to call them.

Using precoded events and user events

PFC includes extensive precoded functionality. This means that by enabling a PFC service, PFC objects detect the enabled service and perform the processing defined in the precoded events.

An example of a precoded event is the u_dw Clicked event, which calls certain DataWindow service functions if they are enabled.

You can extend these events; do not override them.

For information on accessing the return value from an ancestor event, see “Calling ancestor functions and events” on page 45.

Using empty user events

PFC includes empty user events into which you can add application-specific code. Many of these events are triggered by menu items, using the message router. Others are meant to be triggered by application-specific code.

An example of an empty user event is the u_dw pfc_Retrieve event, to which you add logic that retrieves rows:

```
Return this.Retrieve()
```

For complete information on PFC user events, see the *PFC Object Reference*.

How PFC uses events

When using events in the context of services, PFC typically behaves as follows:

1. Within the event on the requestor object, call the corresponding event on the service object, passing arguments as appropriate. For example, the u_dw Clicked event calls the n_cst_dwsrv_sort pfc_Clicked event, passing the x position, y position, row, and DW object (that is, the arguments to the DataWindow Clicked event).
2. The event on the service object performs the required action, calling other object functions as appropriate. For example, the n_cst_dwsrv_sort pfc_Clicked event performs extensive processing, including calls to n_cst_dwsrv_sort functions.

Use the events
Although you can usually call PFC object functions directly, it's easier to call the corresponding events since they already contain error checking.

Using PFC pre-event processes

PFC includes many pre-event processes, to which you add code that customizes or extends the functionality of the associated event. For example, you add code to the pfc_PreRMBMenu event to control the items that appear in a pop-up menu. Other events that feature pre-event processing include:

pfc_PreAbout
pfc_PreClose
pfc_PreLogonDlg
pfc_PreOpen
pfc_PrePageSetupDlg
pfc_PrePrintDlg
pfc_PreRestoreRow
pfc_PreSplash
pfc_PreToolbar
pfc_PreUpdate

Typically, these events are passed an autoinstantiated user object by reference. This user object contains properties used to control processing in the associated event. You modify user object properties to modify or extend processing. In some cases, you will need to modify additional objects. For example, to control the display of an additional field in the About box, you might:

1. Extend the n_cst_aboutattrib user object by adding an instance variable that contains the value to be displayed in the w_about window (a user ID in the example ahead).
2. Add the field to the w_about window (sle_userid in the example ahead).
3. Add code to the w_about Open event that accesses the n_cst_aboutattrib user object (available as the inv_aboutattrib instance variable) and copies the user ID to the SingleLineEdit:

   ```
   sle_userid.text = inv_aboutattrib.is_userid
   ```

4. Add code to the n_cst_appmanager pfc_PreAbout event to initialize the value:

   ```
   anv_aboutattrib.is_userid = this.of_GetUserID()
   ```

To display w_about, call the application manager of_About function.

Using attribute objects

PFC provides a number of attribute-only user objects. These user objects:

- Contain public instance variables
- Are autoinstantiated
- Have names that end with *attrib*
- Are often used to pass information to PFC pre-event processes, such as pfc_PreAbout
- Are extensible (you can define additional instance variables)

Because you can extend these objects, PFC uses them instead of structures.

In addition to defining additional public instance variables, you can also use access levels and object functions to further customize the object’s behavior.

Attribute objects include:

Attribute object	Associated with	Usage
n_cst_aboutattrib	Pfc_PreAbout (n_cst_appmanager)	Open w_about by calling the n_cst_appmanager of_About function
n_cst_calculatorattrib	Constructor (u_calculator)	Internal
n_cst_calendarattrib	Constructor (u_calendar)	Internal
n_cst_columnattrib	ListView data access objects	Set with of_ RegisterReportColumn
n_cst_dberrorattrib	Logical unit of work service (n_cst_luw)	Internal
n_cst_dirattrib	File service objects	Internal
n_cst_dssrv_multitableattrib	DataStore multitable update service	Internal
n_cst_dwcacheattrib	Caching service	Internal
n_cst_dwobjectattrib	Of_Describe (n_cst_dssrv and n_cst_dwsrv)	Of_Describe returns DataWindow properties in this object
n_cst_dwpropertyattrib	DataWindow Properties objects	Internal
n_cst_dwsrv_dropdownsearchattrib	Search service for DropDownDataWindows and DropDownListBoxes	Internal
n_cst_dwsrv_multitableattrib	DataWindow multitable update service	Internal
n_cst_dwsrv_querymodeattrib	Service to enable or disable query mode	Internal
n_cst_dwsrv_resizeattrib	DataWindow resize service	Set with n_cst_dwsrv_ resize of_register function
n_cst_errorattrib	Error message service	Used to pass display information to w_message
n_cst_filterattrib	DataWindow filter service	Used to pass information to filter dialog boxes
n_cst_findattrib	DataWindow find service	Used to pass information to Find dialog box
n_cst_infoattrib	DataWindow Properties objects	Internal
n_cst_itemattrib	PFC ListBox, PictureListBox, and TreeView	Internal
n_cst_linkageattrib	DataWindow linkage service	Internal
n_cst_logonattrib	Pfc_PreLogonDlg (n_cst_appmanager)	Open w_logon by calling the n_cst_appmanager of_LogonDlg function
n_cst_lvsrvattrib	ListView data access objects	Set with of_Register

Attribute object	Associated with	Usage
n_cst_mruattrib	MRU service	Use in the window's pfc_MRUProcess and pfc_PreMRUSave events
n_cst_propertyattrib	DataWindow Properties objects	Internal
n_cst_resizeattrib	Resize service	Internal
n_cst_restorerowattrib	DataWindow row manager service	Internal
n_cst_returnattrib	DataWindow filter and sort services	Internal
n_cst_selectionattrib	Selection service	Populated with arguments to the n_cst_selection of_Open function
n_cst_sortattrib	DataWindow sort service	Used to pass information to the sort dialog boxes
n_cst_splashattrib	Pfc_PreSplash event (n_cst_appmanager)	Open w_splash by calling the n_cst_appmanager of_Splash function
n_cst_sqlattrib	SQL service	Contains the components of a SQL SELECT statement
n_cst_textstyleattrib	PFC RichTextEdit control	Use to get and set text properties (bold, italic, and so on)
n_cst_tmgregisterattrib	Timing service	Internal
n_cst_toolbarattrib	Pfc_PreToolbars event (w_frame)	Open w_toolbars by calling the w_frame pfc_Toolbars event
n_cst_trregistrationattrib	Transaction registration service	Used to track Transaction objects
n_cst_tvattrib	TreeView service	Internal
n_cst_tvsrvattrib	TreeView data access object	Set with of_Register
n_cst_winsrv_sheetmanagerattrib	Sheet management service	Internal
n_cst_winsrv_statusbarattrib	Status bar service	Internal
n_cst_zoomattrib	DataWindow print preview service	Internal

Using PFC constants

Many PFC objects include instance variables that are declared as constants. You can use these instance variables to create more readable code. For example, both of the following functions set the DataWindow linkage style, but the second is easier to understand:

```
// 1 = Filter linkage style.
dw_emp.inv_linkage.of_SetStyle(1)

// FILTER is a constant instance variable
// that is initialized to 1.
dw_emp.inv_linkage.of_SetStyle  &
   (dw_emp.inv_linkage.FILTER)
```

Coding conventions
The PFC convention is to code constants in all caps.

The message router

PFC uses a **message router** to handle communication between menus and windows. This customized message-passing mechanism is built into all PFC menus and windows.

Using the message router

Although you can use the message router to communicate between any object and a window, it is typically used to pass messages from menus to windows. It implements a customized searching algorithm to determine the appropriate object to receive the message.

By using the message router:

- Your menu script only needs to know the user event to call; it doesn't need to know the current window or the associated control name.
- Your windows do not need to maintain user events that simply call DataWindow user events. This reduces the number of user events maintained by the window.

Message = user event
The message passed by a message router function is actually a string containing the name of a user event to be triggered by the window or one of its controls.

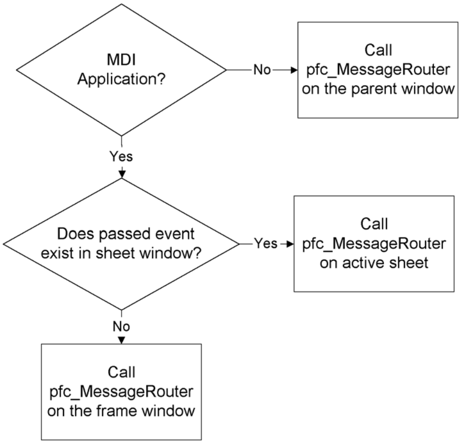
Built-in debugging messages

The message router includes built-in debugging messages to provide error information.

How of_SendMessage works

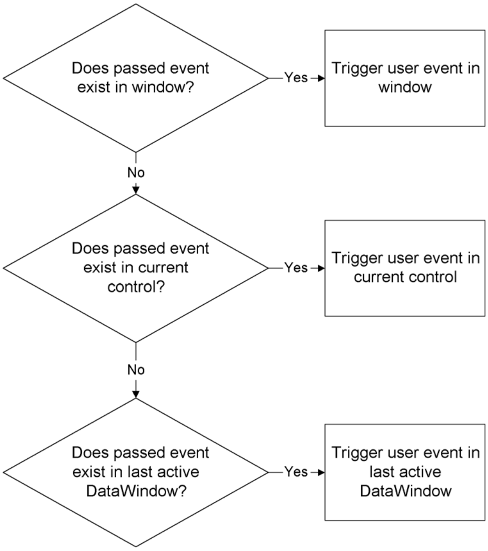
When the user selects a menu item, the item's Clicked event script calls the menu's of_SendMessage function, passing the name of the user event to be called. Of_SendMessage calls the n_cst_menu of_SendMessage function, which calls the window's pfc_MessageRouter event, which in turn calls the specified user event.

Of_SendMessage calls the pfc_MessageRouter user event differently depending on whether the application is MDI or SDI:


MDI Application?
No
Call pfc_MessageRouter on the parent window
Yes
Does passed event exist in sheet window?
Yes
Call pfc_MessageRouter on active sheet
No
Call pfc_MessageRouter on the frame window

How pfc_ MessageRouter works

The pfc_MessageRouter user event calls the passed user event in the window, the active control, and the last active DataWindow:


Does passed event exist in window?
Yes
Trigger user event in window
No
Does passed event exist in current control?
Yes
Trigger user event in current control
No
Does passed event exist in last active DataWindow?
Yes
Trigger user event in last active DataWindow

Pass messages between menus and windows
The message router is primarily a mechanism to communicate between menus and windows. Except for CommandButtons inside DataWindows, you cannot use buttons to call the pfc_MessageRouter event. This is because the message routine calls the GetFocus event to access the current control, which, after you click a CommandButton, is the button itself.

Transaction management with PFC

One of PowerBuilder's key strengths is its ability to access a variety of DBMSs quickly and easily. PowerBuilder uses the transaction object as a communications area between PowerScript and the database. SQLCA is the default PowerBuilder Transaction object.

The n_tr user object

PFC includes the **n_tr** user object. N_tr is a customized Transaction object that inherits from the Transaction system object. This customized Transaction object includes instance variables, events, and functions to encapsulate and extend database communication.

N_tr helps you manage transactions by providing a standard set of functions for performing database connects, disconnects, commits, and rollbacks. Use n_tr functions instead of native SQL transaction management statements. For example, to connect to the database, use of_Connect instead of the CONNECT statement.

Two ways to use n_tr

You use n_tr in two ways:

- **As a replacement for SQLCA** Use the Application painter's Properties dialog box to specify that the default SQLCA will be of data type n_tr
- **In addition to SQLCA** Define an instance variable of type n_tr and create it programmatically

If your application requires more than one Transaction object, you will use both of these methods.

If using more than one Transaction object, you can use the transaction registration service to perform functions such as committing all open transactions or rolling back all open transactions.

See Chapter 4, "Using PFC Services".

v **To associate n_tr with SQLCA:**

1 Access the Application painter.

2 Display the Properties view, click the Additional Properties button and select the Variable Types tab.

3 Type n_tr in the SQLCA box.

4 Click OK.

v **To use n_tr:**

1 If you are using a Transaction object other than SQLCA, create it.

This example assumes an itr_security instance variable of type n_tr.

```
itr_security = CREATE n_tr
```

2 Initialize the ib_autorollback instance variable, which specifies what to do if the application closes (or the object is otherwise destroyed) while the transaction is still connected:

```
itr_security.of_SetAutoRollback(FALSE)
```

Initialize ib_autorollback in the extension level
You can enforce transaction consistency by initializing ib_autorollback in the n_tr Constructor event.

3 Initialize Transaction object fields using the of_Init function:

```
String   ls_inifile

ls_inifile = gnv_app.of_GetAppIniFile()
IF SQLCA.of_Init(ls_inifile,"Database") = -1 THEN
      MessageBox("Database",  &
         "Error initializing from " + ls_inifile)
      HALT CLOSE
END IF
```

4 Connect to the database by calling the of_Connect function:

```
IF SQLCA.of_Connect() = -1 THEN
      MessageBox("Database", &
         "Unable to connect using " + ls_inifile)
      HALT CLOSE
```

```
ELSE
      gnv_app.of_GetFrame().SetMicroHelp &
         ("Connection complete")
END IF
```

5 Call n_tr functions as needed.

Calling ancestor functions and events

In extending ancestor functions and events, you may need to call the ancestor method and continue processing based on its return value. This is especially important when extending PFC events (those that begin with the pfc_ prefix) that use return codes. You must check the return code to ensure that ancestor processing succeeded before performing descendent processing.

Overriding ancestor events
To extend a PFC event that uses a return code, you must override the event and call the ancestor event explicitly, as shown in this discussion.

Use the following syntax to call an ancestor event, passing arguments and receiving a return code:*result* = Super::Event *eventname* (*arguments* ...)

Use the following syntax to call an ancestor function, passing arguments and receiving a return code:*result* = Super::Function *functionname* (*arguments* ...)

This example overrides the u_dw pfc_Update event, writing to an update log if the ancestor event processes successfully:

```
Integer  li_return

// Call ancestor event, passing
// descendant's arguments.
li_return = Super::Event pfc_Update &
      (ab_accepttext, ab_resetflag)
IF li_return = 1 THEN
   // ue_WriteLog is a user-defined event.
      li_return = this.Event ue_WriteLog
END IF
Return li_return
```

Adding online Help to an application

Online Help is an important part of any application. PFC provides functions and events to enable online Help in your application.

For information on PFC dialog Help, see “Deploying PFC dialog box Help” on page 230.

v **To enable online Help in a PFC application:**

1 Within n_cst_appmanager or a descendant, you can use the Properties view to assign the complete name of the Help file to the is_helpfile instance variable.

Alternatively, you can call the of_SetHelpFile function to establish the Help filename. You usually do this in the Constructor event:

```
this.of_SetHelpFile("c:\eis\eisapp.hlp")
```

2 Specify the Help topic associated with the window. The pfc_PreOpen event is a good place for this:

```
Long  ll_helpid

ll_helpid = 1020  // 1020 is a Help topic ID
ia_helptypeid = ll_helpid
```

This allows you to provide detailed online Help for selected windows. You can set ia_helptypeid to either a long (which PFC interprets as a Help topic ID) or a string (which PFC interprets as a search keyword).

3 (Optional) If you are not using a descendant of PFC’s m_master menu, add calls to the window’s pfc_Help user event in your menu’s Help menu items. Pfc_Help is defined in w_master so it is available in all PFC windows.

4 For dialog boxes, call the pfc_Help user event in the Help button’s Clicked event:

```
Parent.Event pfc_Help( )
```

PFC handles window-level Help automatically
The message router calls the active window’s pfc_Help user event when the user selects Help>Help Topics from the menu bar of a menu descended from m_master.

Installing PFC upgrades

Sybase distributes regular maintenance releases between major PowerBuilder releases. In addition to PowerBuilder updates, each maintenance release also includes updates to PFC. The way you apply PFC maintenance depends on your PFC usage:

- **No modifications to either PFC level** If there is no modification to either the PFC ancestor level or the PFC extension level, you can simply install the new set of PBLs over the existing PBLs

  Always make a backup copy
  Always make a backup copy of all PFC PBLs before installing updated PBLs. These instructions assume that you have made a backup.

- **One or more intermediate extension levels or developer code in the PFC extension level** If you have changed any of the levels below the PFC ancestor level, you must ensure that extensions and other modifications are not overwritten, as described in the discussion below.

v **To upgrade to the latest PFC release:**

1 Move all extension-level PBLs to a directory that will not be overwritten by the install procedure.

PFC ancestor objects
You should never modify PFC ancestor objects (objects with the pfc_ prefix). These instructions assume no modifications have been made to PFC ancestor objects.

2 Determine your current version. You can find the current version at the top of the current PFC *readme.txt* file or in instance variables defined in pfc_n_cst_debug. The version is in the format *majorrevision.minorrevision.fixesrevision.*

3 Run the install procedure, placing the PFC PBLs in the current PFC directory and overwriting the current PFC ancestor PBLs.

4. Merge existing extension objects with new extension objects. Review the newly installed *readme.txt* file to see a list of new extension objects. There are two methods of merging existing extension objects with new extension objects:

   - **Copy new objects to customized extension PBLs** Copy each new object from the newly installed PFC extension level PBL to your customized extension PBL. Then copy the customized extension PBLs back to their original directory, overwriting the newly installed PFC extension PBLs.

   - **Copy existing objects to the new PFC extension PBL** Copy all objects from the customized extension PBLs to the appropriate newly installed PFC extension PBL.

5. Start PowerBuilder.

6. Adjust the application target library list if necessary.

7. Perform a full rebuild of the target.

CHAPTER 4 Using PFC Services

About this chapter

This chapter explains PFC services and how to use them.

Contents

Application services

PFC provides the following application services:

DataWindow caching
Debugging
Error
Application preference
Most recently used object

Security
Transaction registration

You control application services through n_cst_appmanager, the application manager. Use application manager functions to enable and disable application services. Because they are scoped to the application manager, which you define as a global variable, application services are available from anywhere within your application.

DataWindow caching service

Overview

The DataWindow caching service buffers data for DataWindow objects. By keeping rows in memory, the DataWindow caching service helps to reduce database access, optimizing application performance. The DataWindow caching service supports the following data sources:

- DataWindow object (using either data retrieved from the database or data stored with the DataWindow object)
- SQL statement
- DataWindow control
- DataStore control
- Rows from an array
- A file

The DataWindow caching service uses PowerBuilder DataStores to buffer data.

PFC enables DataWindow caching through the n_cst_dwcache user object.

PFC code is in ancestor-level objects
This book always refers to extension-level objects (such as n_cst_dwcache). All PFC code is actually in ancestor-level objects (such as pfc_n_cst_dwcache).

Usage

Use DataWindow caching to minimize database access and optimize performance.

v **To enable DataWindow caching:**

- Call the n_cst_appmanager of_SetDWCache function:

```
gnv_app.of_SetDWCache(TRUE)
```

v **To use DataWindow caching:**

1. Cache data by calling the of_Register function, passing different arguments depending on the data to be cached:

   - To cache rows retrieved from the database via a DataWindow object, pass an identifier, a Transaction object, the DataWindow object name, and arguments if any
   - To cache rows retrieved from the database via a SQL statement, pass an identifier, a Transaction object, and the SQL statement
   - To cache rows in an array, pass an identifier, the DataWindow object name, and the data
   - To cache rows from a DataWindow control, pass an identifier and the DataWindow control
   - To cache rows from a DataStore, pass an identifier and the DataStore instance
   - To cache rows from a file, pass an identifier and the filename

2. To determine if a DataWindow object is already registered with the caching service, call the of_IsRegistered function, passing the object's identifier.

3. To access cached data from, call the of_GetRegistered function. This example assumes an ids_datastore instance variable:

   ```
   gnv_app.inv_dwcache.of_GetRegistered  &
         ("d_emplist", ids_datastore)
   ids_datastore.ShareData(dw_emplist)
   ```

4. To re-retrieve data for a cached DataWindow, call the of_Refresh function.

5. To stop caching, call the of_UnRegister function.

6. (Optional) Disable the DataWindow caching service by calling the n_cst_appmanager of_SetDWCache function:

   ```
   gnv_app.of_SetDWCache(FALSE)
   ```

   In most cases, you do not disable DataWindow caching explicitly. PFC destroys n_cst_dwcache automatically when your application shuts down.

Debugging service

Overview

The debugging service automatically displays messages when PFC encounters conditions that indicate an error.

The PFC message router uses the debugging service to control the display of error messages when a passed event does not exist.

Development tool only
The PFC debugging service is a development tool only. Do not enable it in production applications.

Usage

Use the debugging service to help you solve problems in the PFC development environment.

v **To use the debugging service:**

1 Enable the debugging service by calling the n_cst_appmanager of_SetDebug function:

```
gnv_app.of_SetDebug(TRUE)
```

2 PFC objects check for the target’s debugging status and, in certain conditions, display error messages.

3 (Optional) Disable the debugging service by calling the n_cst_appmanager of_SetDebug function:

```
gnv_app.of_SetDebug(FALSE)
```

In most cases, you do not disable the debugging service explicitly.

For information on SQL Spy and the DataWindow Property window (two debugging utilities supplied with PFC) see Chapter 7, “PFC Utilities”.

Application preference service

Overview

You use the application preference service to save and restore application and user information using either an INI file or the Windows registry. Saving and loading application settings has two advantages:

- **Persistence** By saving application state, users don’t have to reset their application preferences each time they start the application
- **Ease of maintenance** By externalizing application settings, you can update application settings without updating code in n_cst_appmanager

PFC enables the application preference service through the n_cst_apppreference user object.

This service saves the following application information:

User key
MicroHelp
Help file
Version
Logo bitmap
Copyright notice
DDETimeOut property
DisplayName property
DWMessageTitle property
MicrohelpDefault property
RightToLeft property
ToolbarFrameTitle property
ToolbarPopMenuText property
ToolbarSheetTitle property
ToolbarUserControl property

The application preference service can also save the following user information:

ToolbarText property
ToolbarTips property
User ID

Saving and loading settings

The application preference service automatically loads settings when the application opens and stores them when the application closes. This information is stored in either the registry (available on Windows) or an INI file (available on all platforms), which you specify as follows:

- **Registry** Call the of_SetUserKey function, specifying the registry key that contains application preference information.
- **INI file** Call the of_SetUserINIFile function, specifying the INI file that contains application preference information.

v **To use the application preference service:**

1 Enable the application preference service by calling the n_cst_appmanager of_SetAppPref function:

```
gnv_app.of_SetAppPref(TRUE)
```

2. Specify the platform-specific repository for application preferences. This example from an application manager Constructor event saves application preferences in the registry or INI file, depending on the execution platform. It assumes you've already established n_cst_appmanager specifications for application key, user key, application INI file, and user INI file:

   ```
   IF this.of_IsRegistryAvailable() THEN
          this.inv_apppref.of_SetAppKey  &
         (this.of_GetAppKey())
          this.inv_apppref.of_SetUserKey  &
         (this.of_GetUserKey())
   ELSE
          this.inv_apppref.of_SetAppINIFile  &
         (this.of_GetAppINIFile())
          this.inv_apppref.of_SetUserINIFile  &
         (this.of_GetUserINIFile())
   END IF
   ```

3. Specify the types of information to save by calling the of_SetRestoreApp and of_SetRestoreUser functions:

   ```
   this.inv_apppref.of_SetRestoreApp(TRUE)
   this.inv_apppref.of_SetRestoreUser(TRUE)
   ```

Most recently used object service

Overview

You use the most recently used (MRU) object service to display a list of most recently used windows on the File menu. By default this list displays up to five items, but, you can increase this number.

PFC enables the MRU service through the n_cst_mru user object.

The MRU service automatically loads MRU information when the application opens. The service saves information in either the registry (available on Windows) or an INI file (available on all platforms), which you specify as follows:

- **Registry** Call the of_SetUserKey function, specifying the registry key that contains MRU information
- **INI file** Call the of_SetUserINIFile function, specifying the INI file that contains MRU information

You must write processing code

To use the MRU service, you must extend the following window events in all windows that are to display on the file menu as MRU objects:

- **Pfc_MRUProcess** Add code that uses the passed MRU information to open the specified window
- **Pfc_PreMRUSave** Add code that saves MRU information
- **Pfc_MRURestore** Add code that restores MRU information

Use IDs to identify groups of windows

You specify IDs to identify windows or groups of windows that appear together on the file menu. By using IDs you can restrict and customize MRU display. For example, when displaying a particular sheet, you might want to restrict MRU display to instances of that sheet only. In other applications, you might want all sheets to display the same MRU items.

Integration with PFC menus

The PFC m_master menu includes five MRU items at the end of the File menu. You can add more MRU items if necessary.

If your application uses non-PFC menus, use m_master as a model in creating your own MRU menu items.

v **To use the MRU service:**

1 Enable the MRU service by calling the n_cst_appmanager of_SetMRU function:

```
gnv_app.of_SetMRU(TRUE)
```

2 Specify where MRU information is to be saved by calling either the n_cst_mru of_SetUserKey function (on Windows platforms) or the of_SetUserINIFile function (all platforms). This example from an application manager Constructor event saves MRU information in the registry or INI file. It assumes you've already established the n_cst_appmanager user key or user INI file:

```
IF this.of_IsRegistryAvailable() THEN
         this.inv_mru.of_SetUserKey  &
        this.of_GetUserKey())
ELSE
         this.inv_mru.of_SetUserINIFile  &
        (this.of_GetUserINIFile())
END IF
```

3 Register IDs to be tracked by the MRU service by calling the n_cst_mru of_Register function. (An ID is the identifier that the window uses to retrieve information through the MRU service.) This is an example of code you can add to the pfc_PreOpen event of the MDI frame window:

```
IF IsValid(gnv_app.inv_mru) THEN
      gnv_app.inv_mru.of_Register("myapp")
END IF
```

4 Extend the pfc_MRUProcess event in each window that uses MRU processing, adding code to open the window or sheet passing the necessary arguments (be sure to add similar code to the frame window if you want to specify MRU items on the frame menu):

```
Window lw_frame, lw_window
n_cst_menu lnv_menu
n_cst_mruattrib lnv_mruattrib

// Check parameters.
IF IsNull(ai_row) THEN
      Return -1
END IF
IF NOT IsValid(gnv_app.inv_mru) THEN
      Return -1
END IF
// Retrieve row from DataStore.
gnv_app.inv_mru.of_GetItem &
   (ai_row, lnv_mruattrib)
// Get the MDI frame, if necessary.
lnv_menu.of_GetMDIFrame(this.menuid, lw_frame)
OpenSheet(lw_window, &
   lnv_mruattrib.is_classname, lw_frame)
Return 1
```

Performing other actions in the pfc_MRUProcess event
To see other types of processing you can perform in the pfc_MRUProcess event, see the comments in the pfc_w_master pfc_MRUProcess event.

5 Extend the pfc_PreMRUSave event in each window that uses the MRU service. In this event, populate the n_cst_mruattrib object with the ID, classname, key, item text, and MicroHelp to be saved:

```
anv_mruattrib.is_id = "myapp"
anv_mruattrib.is_classname = this.ClassName()
anv_mruattrib.is_menuitemname = this.Title
anv_mruattrib.is_menuitemkey = this.ClassName()
anv_mruattrib.is_menuitemmhelp =  &
   "Opens " + this.Title
Return 1
```

6 Extend the pfc_MRURestore event in each window that uses the MRU service. In this event, set the ID of the information you want to display on the menu:

```
If IsValid(gnv_app.inv_mru) Then
   Return gnv_app.inv_mru.of_Restore("myapp", This)
End If
```

7 Call the pfc_MRUSave event to save MRU information. You can call this event when the window opens, when information is saved, or when the window closes (this example is from the pfc_PreOpen event):

```
this.Event pfc_MRUSave()
```

Error message service

Overview

The error message service provides many features for displaying and logging your application's error messages. You can display messages in either the PowerBuilder MessageBox or in the PFC w_message dialog box. Both display options offer the following features:

- **Message logging** Message logging to a file, including multiplatform support. PFC automatically logs messages whose severity is greater than a specified level
- **MAPI support** Automatic error notification via e-mail (MAPI-compliant e-mail systems only). PFC automatically sends e-mail notification for messages whose severity is greater than a specified level
- **Message database** Access to a database of predefined messages (which can reside in either a database or a file). Predefined messages provide standardization of message text, elimination of duplicate messages, and ease of localization
- **Symbolic parameter replacement** Messages can have arguments that are replaced at execution time
- **Unattended option** Messages are logged (or e-mailed) but do not display

If you use the w_message dialog box, you have additional options:

- **User input and print buttons** The user can print messages and can optionally add comments (this is especially useful when logging messages and when using the automatic e-mail notification feature)
- **Automatic close** The w_message dialog box will close automatically after a specified number of seconds

W_message bitmaps
If you use the w_message dialog box, the bitmaps it uses must be available at execution time.

Usage

Use the error service to handle all of your application's message and error handling. If you are keeping messages in a database, you can either use the messages tables in PFC.DB or pipe it to your application's database.

Using the messages table
In most cases you should copy the messages table to your application's database. This table contains predefined PFC error messages as well as those that you define.

v **To use the error message service:**

1 Create an instance of n_cst_error by calling the n_cst_appmanager of_SetError function (this example is from an n_cst_appmanager pfc_Open event):

```
this.of_SetError(TRUE)
```

2 Specify the error message source:

- If the source is a file, call the following function:

```
this.inv_error.of_SetPredefinedSource &
        ("c:\eisapp\eiserr.txt")
```

- If the source is a database, call the following function:

```
this.inv_error.of_SetPredefinedSource &
        (itr_error)
```

The error message source
When using a file as the error message source, the file must contain all rows found in the PFC.DB messages table; columns must be delimited by tabs.

PFC uses predefined messages in certain situations. If you enable the error message service and receive message display errors, make sure the error message source has been established correctly.

Additional user-defined messages must conform to the format of the messages table (also used by the d_definedmessages DataWindow object).

3 (Optional) Specify the name of the log file (to disable logging, call of_SetLogFile passing an empty string):

```
this.inv_error.of_SetLogFile &
       ("c:\workingdir\errlog.txt")
```

4 (Optional) Specify the user ID (used in message logging):

```
this.inv_error.of_SetUser &
 (this.of_GetUserID())
```

5 (Optional) Specify the types of messages for which the error service will provide automatic notification and logging:

```
this.inv_error.of_SetNotifySeverity(5)
this.inv_error.of_SetLogSeverity(4)
```

6 (Optional) If your application uses the error service's automatic notification feature, specify the current user's e-mail ID and password. Also specify the e-mail IDs of the users to be notified automatically. This example assumes a mechanism for storing e-mail IDs and user passwords:

```
this.inv_error.of_SetNotifyConnection &
       (ims_mailsess)
this.inv_error.of_SetNotifyWho(is_autonotify)
```

N_cst_appmanager pfc_Open
The steps listed above can all be coded in the n_cst_appmanager pfc_Open event.

7. In your application error checking, call the of_Message function to display messages, with optional logging and notification. The of_Message function allows you to either use the message database or specify message text dynamically. This example uses the message database:

   ```
   gnv_app.inv_error.of_Message &
   ("EIS0210")
   ```

v **To use symbolic parameters (predefined messages only)**

1. Define messages in the messages table. Type % to mark the places to be replaced at runtime with symbolic parameters. For example:

   ```
   EIS1030   Unable to find the file % in %
   ```

2. Create an array of replacement arguments:

   ```
   String  ls_parms[ ]

   ls_parms[1] = "logfile.txt"
   ls_parms[2] = "c:\windows\system"
   ```

3. Call of_Message, passing the array:

   ```
   gnv_app.inv_error.of_Message("EIS1030", ls_parms)
   ```

   PFC displays the message, replacing the first % with the first element in the ls_parms array and the second % with the second element in the ls_parms array.

Security service

Overview

PFC's security feature can handle many of your application's security needs. It includes administrative components and a runtime security object, n_cst_security.

Usage

To use the PFC security system, you must first define users and groups, associate them with windows, menus, user objects, and controls, and then add code to your application.

v **To use the security service:**

1. Define users and groups, as described in Chapter 7, "PFC Utilities".
2. Define security for your application's window controls, menus, user objects, and controls, as described in Chapter 7, "PFC Utilities".

3 Create the security object by calling the n_cst_appmanager of_SetSecurity function (this example is from an n_cst_appmanager pfc_Open event):

```
this.of_SetSecurity(TRUE)
```

4 Establish a Transaction object for the security database. This example assumes an itr_sec instance variable of type n_tr on n_cst_appmanager:

```
itr_sec = CREATE n_tr
CONNECT using itr_sec;
```

5 Initialize the security object by calling the of_InitSecurity function:

```
this.inv_security.of_InitSecurity  &
       (itr_sec, "EISAPP", &
   gnv_app.of_GetUserID(), "Default")
```

N_cst_appmanager pfc_Open
The steps listed above can all be coded in the n_cst_appmanager pfc_Open event.

6 Disconnect from the database and destroy the Transaction object when the application closes. This example might be coded in the n_cst_appmanager pfc_Close event:

```
DISCONNECT using itr_sec;
Destroy itr_sec
```

7 In the Open or pfc_PreOpen events of windows for which you want to apply security, call the of_SetSecurity function:

```
IF NOT &
gnv_app.inv_security.of_SetSecurity(this) THEN
   MessageBox("Security", &
      "Unable to set security")
   Close(this)
END IF
```

Other places to call of_SetSecurity
You might also call the n_cst_security of_SetSecurity function from the Constructor event of a DataWindow, visual user object, or menu for which you want to implement security.

Transaction registration service

Overview

The transaction registration service tracks the transaction objects used by your application. This service is for use with Transaction objects based on n_tr.

PFC enables transaction registration through the n_cst_trregistration user object.

Usage

Use this service to keep track of transactions when your application uses more than one transaction.

When your application closes, this object automatically destroys all open registered transactions. Set the n_tr ib_autorollback instance variable to TRUE to cause closing transactions to COMMIT; set ib_autorollback to FALSE to cause a ROLLBACK. You set this instance variable with the n_tr of_SetAutoRollback function.

v **To enable the transaction registration service:**

- Call the n_cst_appmanager of_SetTrRegistration function:

```
gnv_app.of_SetTrRegistration(TRUE)
```

The application manager destroys the transaction registration service automatically when the application closes.

v **To register a transaction:**

- Call the n_cst_trregistration of_Register function:

```
gnv_app.inv_trregistration.of_Register(SQLCA)
```

v **To control whether the transaction registration service commits or rolls back open transactions when it is destroyed:**

- Call the n_tr of_SetAutoRollback function:

```
SQLCA.of_SetAutoRollback(TRUE)
```

If you set autorollback to TRUE and the object is still connected, the service rolls back open transactions when it is destroyed; if you set it to FALSE, it commits open transactions. However, to ensure that transactions close properly, your application should issue COMMITs, ROLLBACKS, and DISCONNECTs explicitly.

v **To establish a transaction name:**

- Call the n_tr of_SetName function:

```
itr_security.of_SetName("Security")
```

v **To close all transactions explicitly:**

1 In the application manager pfc_Close event (or some other appropriate place), call the n_cst_trregistration of_GetRegistered function:

```
n_tr   ltr_trans[]
Integer  li_max, li_count

li_max =  &
   this.inv_trregistration.of_GetRegistered  &
     (ltr_trans)
```

2 Loop through the n_tr array, committing and destroying transactions as appropriate:

```
FOR li_count = 1 to li_max
COMMIT using ltr_trans[li_count];
DESTROY ltr_trans[li_count]
NEXT
```

DataWindow services

Most production-strength PowerBuilder applications make intense use of DataWindow controls. PFC provides a wide variety of DataWindow services that you can use to add production-strength features to an application. Many of these services require little or no coding on your part.

PFC implements DataWindow services through a set of custom class user objects that descend from a common ancestor. The ancestor object contains functions, events, and instance variables that are required by multiple services. Each DataWindow service contains additional functions, events, and instance variables.

Accessing DataWindow services

To access DataWindow services, you create DataWindow objects that are based on the u_dw user object. U_dw contains:

- Functions to enable and disable DataWindow services
- Instance variables that allow you to reference each service's functions, events, and instance variables (this type of instance variable is called a **reference variable**)

- Precoded events and user events that call the DataWindow service's functions and events
- Empty user events to which you add code to perform application-specific processing

Use u_dw for all DataWindow controls
Use the u_dw user object for all of your application's DataWindow controls.

Enabling DataWindow services

Each DataWindow control enables only the required DataWindow services. This minimizes application overhead.

The following table lists DataWindow services and how they are implemented:

DataWindow service	Implementation
Basic DataWindow service (ancestor for all other services)	n_cst_dwsrv
Drop-down search service	n_cst_dwsrv_dropdownsearch
Filter service	n_cst_dwsrv_filter
Find and replace service	n_cst_dwsrv_find
Linkage service	n_cst_dwsrv_linkage
Multitable update service	n_cst_dwsrv_multitable
Print preview service	n_cst_dwsrv_printpreview
DataWindow property service	n_cst_dwsrv_property
Querymode service	n_cst_dwsrv_querymode
Reporting service	n_cst_dwsrv_report
Required column service	n_cst_dwsrv_reqcolumn
DataWindow resize service	n_cst_dwsrv_resize
Row management service	n_cst_dwsrv_rowmanager
Row selection service	n_cst_dwsrv_rowselection
Sort service	n_cst_dwsrv_sort

DataWindow services ancestor

Overview

The DataWindow services ancestor contains instance variables, events, and functions for use by all other DataWindow services. You can use many of the ancestor functions too.

PFC enables basic DataWindow services through the n_cst_dwsrv user object.

DataStore services
This service is available to the n_ds DataStore via the n_cst_dssrv user object.

Usage

Use this service for general DataWindow functionality, including:

- Getting and setting DataWindow information
- As an alternative to the Modify and Describe PowerScript functions
- DataWindow service defaults

Ancestor functions are available in the descendants
Because the n_cst_dwsrv user object is the ancestor for all DataWindow services, its functions are also available through any of the descendent DataWindow service user objects.

v **To enable basic DataWindow services:**

- Call the u_dw of_SetBase function:

```
dw_emplist.of_SetBase(TRUE)
```

U_dw destroys the service automatically when the DataWindow is destroyed.

v **To access DataWindow information:**

- Call one of the following n_cst_dwsrv functions:

Function	When to call
of_Describe	To access information on DataWindow attributes and columns
of_GetHeaderName	To determine the header name for a specified DataWindow column
of_GetHeight	To determine a column’s height
of_GetObjects	To access the names of the objects within a DataWindow
of_GetWidth	To determine a column’s width
of_GetItem of_GetItemAny	To retrieve data for a DataWindow column, regardless of data type

v **To set DataWindow data:**

- Call one of the following n_cst_dwsrv functions:

Function	When to call
of_Modify	To set DataWindow attributes and columns
of_SetItem	To set or modify the display value for a DataWindow column, regardless of datatype

v **To refresh all DropDownDataWindows in a DataWindow:**

- Call the of_PopulateDDDWs function:

```
Integer li_return

li_return = &
      dw_emplist.inv_base.of_PopulateDDDWs()
gnv_app.of_GetFrame().SetMicroHelp &
      (String(li_return) + " DDDW columns
refreshed")
```

v **To access DataWindow service defaults:**

- Call one of the following n_cst_dwsrv functions:

Function	When to call
of_GetColumnDisplayName	To determine when DataWindow services display when referring to columns
of_GetColumnNameStyle	To determine what DataWindow services display when referring to columns
of_GetDefaultHeaderSuffix	To determine the default DataWindow suffix for header columns
of_GetDisplayItem of_GetDisplayUnits	To determine the text displayed when displaying CloseQuery message
of_SetColumnDisplayNameStyle	To specify what DataWindow services display when referring to columns: • DataWindow column names • Database column names • DataWindow column header names
of_SetDefaultHeaderSuffix	To specify the default DataWindow suffix for header columns (_t is the default)
of_SetDisplayItem of_SetDisplayUnits	To specify the text displayed when displaying CloseQuery message

Drop-down DataWindow search service

Overview

The PFC drop-down DataWindow search service automatically scrolls drop-down DataWindows to items that begin with the typed letter. For example, when a user types S in a drop-down DataWindow, this service automatically scrolls the list to the first item that begins with S. If the user then types A, the service scrolls to the first item that begins with A.

PFC enables the drop-down DataWindow search service through the n_cst_dwsrv_dropdownsearch user object.

Usage

You establish drop-down DataWindow search functionality by enabling the service and adding code to two DataWindow events.

v **To enable the drop-down DataWindow search service:**

1 Call the u_dw of_SetDropDownSearch function:

```
this.of_SetDropDownSearch(TRUE)
```

U_dw destroys the service automatically when the DataWindow is destroyed.

2 In the DataWindow control’s EditChanged event, add a call to the n_cst_dropdownsearch pfc_EditChanged event:

```
this.inv_dropdownsearch.Event pfc_EditChanged &
      (row, dwo, data)
```

3 In the DataWindow control’s ItemFocusChanged event, add a call to the n_cst_dwsrv_dropdownsearch pfc_ItemFocusChanged event:

```
this.inv_dropdownsearch.Event &
      pfc_ItemFocusChanged(row, dwo)
```

4 Specify the DropDownDataWindow column for which the service is enabled by calling the of_AddColumn function:

```
this.inv_dropdownsearch.of_AddColumn("dept_id")
```

Filter service

Overview

The PFC filter service allows you to provide easy-to-use filter capabilities in a DataWindow.

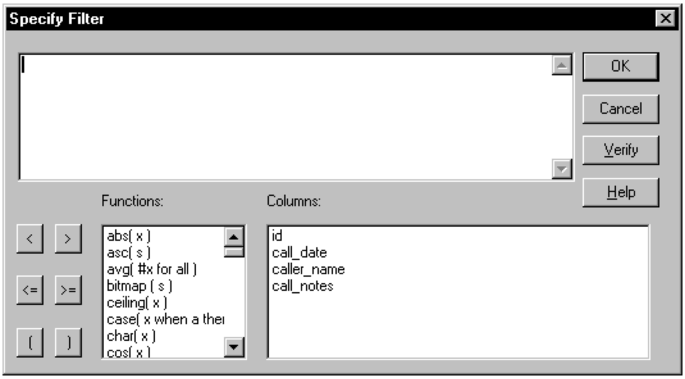
Use this service to add filter capabilities to your application.

PFC enables the filter service through the n_cst_dwsrv_filter user object.

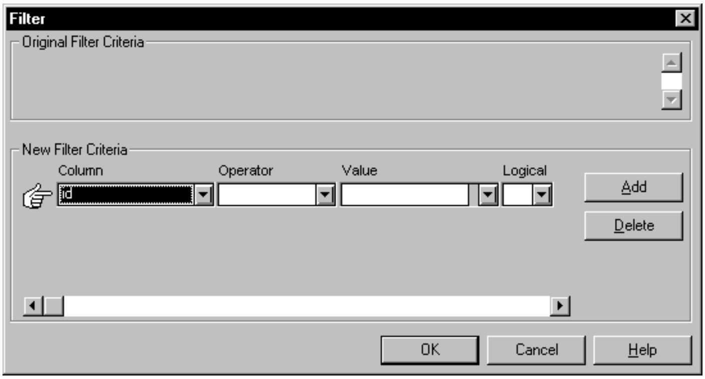
Usage

The filter service displays Filter dialog boxes automatically. All you do is enable the service and specify the filter style you want. You can choose among three styles of filter dialog boxes:

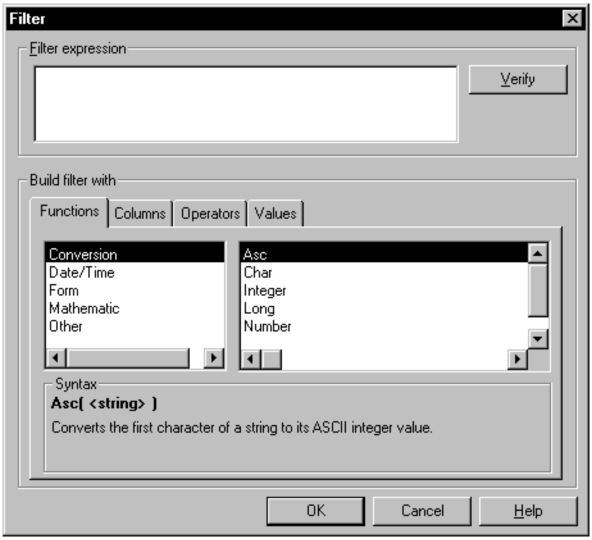
- Default PowerBuilder Filter dialog box:



- Drop-down list box interface (w_filtersimple):



- Tabbed interface (w_filterextended):



Asc(<string>)

Converts the first character of a string to its ASCII integer value.

v **To enable the filter service:**

- Call the u_dw of_SetFilter function, set the Transaction object, and specify that Filter dialog boxes use DataWindow column header names:

```
dw_emp.of_SetFilter(TRUE)
dw_emp.of_SetTransObject(SQLCA)
dw_emp.inv_filter.of_SetColumnDisplayNameStyle &
    (dw_emp.inv_filter.HEADER)
```

Filtering by column header
If you filter by column header, make sure that all columns added to the DataWindow have headers, and that these conform to the naming scheme for headers. The default naming scheme uses the suffix _t, but you can change this by calling the of_SetDefaultHeaderSuffix function.

U_dw destroys the service automatically when the DataWindow is destroyed.

v **To specify the filter style:**

- Call the of_SetStyle function, specifying the Filter dialog box type:

```
dw_emplist.inv_filter.of_SetStyle  &
    (dw_emp.inv_filter.SIMPLE)
```

v **To display the filter dialog box:**

- Call the pfc_FilterDlg event:

```
dw_emplist.inv_filter.Event pfc_FilterDlg( )
```

You do not typically call this event. In most cases, the user displays the Filter dialog box by selecting View>Filter from the menu bar.

Find and replace service

Overview

The PFC find service allows you to add find and replace functionality to your application's DataWindows.

PFC enables the find and replace service through the n_cst_dwsrv_find user object.

Usage

You can use the find service to add find and replace functionality to DataWindows, displaying either the w_find dialog box or the w_replace dialog box. These boxes display automatically if you've enabled the find service and the user selects Edit>Find or Edit>Replace from the menu bar of a menu that descends from PFC's m_master menu.

v **To enable the find service:**

- Call the u_dw of_SetFind function:

```
dw_emplist.of_SetFind(TRUE)
```

U_dw destroys the service automatically when the DataWindow is destroyed.

v **To display the w_find dialog box:**

- Call the u_dw pfc_FindDlg event:

```
dw_emplist.Event pfc_FindDlg()
```

You do not typically call this event. In most cases, the user displays the w_find dialog box by selecting Edit>Find from the menu bar.

v **To display the w_replace dialog box:**

- Call the u_dw pfc_ReplaceDlg event:

```
dw_emplist.Event pfc_ReplaceDlg()
```

You do not typically call this event. In most cases, the user displays the w_replace dialog box by selecting Edit>Replace from the menu bar.

Linkage service

Overview

The PFC linkage service helps you to create master/detail windows and other types of windows that require coordinated processing.

The linkage service contains the following features:

- **Linkage style** Controls whether detail DataWindows retrieve rows, filter rows, or scroll to the appropriate row
- **Update style** Controls how the linkage service updates DataWindows (top-down, bottom-up, top-down then bottom-up, bottom-up then top-down, or a developer-specified custom update)
- **Confirm on row change** When the master changes rows, this option displays a confirmation dialog if modifications made to detail DataWindows will be lost
- **Confirm on delete** Displays a confirmation dialog when the user deletes rows
- **Cascading key changes** The linkage service automatically updates detail DataWindows when you change a key value on the master
- **Delete style** When you delete a master row, this option specifies whether the linkage service deletes detail rows, discards detail rows, or leaves them alone
- **Extended update** Allows you to integrate other controls (such as ListViews, TreeViews, and DataStores) into the default save process

The linkage service is completed integrated with n_cst_luw and with the w_master pfc_Save process.

Sharing data between DataWindows
You can use the PowerScript ShareData function to share data between master and detail DataWindows. However, do not insert rows into the detail DataWindow when sharing data.

PFC enables the linkage service through the n_cst_dwsrv_linkage user object.

Usage

You can use the linkage service to coordinate any type of processing among DataWindows. However, the most common use is for master/detail processing.

v **To enable the linkage service:**

- Call the u_dw of_SetLinkage function:

  ```
  dw_emplist.of_SetLinkage(TRUE)
  ```

  U_dw destroys the service automatically when the DataWindow is destroyed.

v **To use the linkage service to coordinate Master/Detail processing:**

1. Enable the linkage service for both the master and detail DataWindows by calling the u_dw of_SetLinkage function, once for each DataWindow:

   ```
   dw_master.of_SetLinkage(TRUE)
   dw_detail.of_SetLinkage(TRUE)
   ```

2. Call the u_dw of_SetTransObject function to establish the Transaction object for the master and detail DataWindows:

   ```
   dw_master.inv_linkage.of_SetTransObject(SQLCA)
   ```

3. Link the detail to the master by calling the of_SetMaster function in the detail DataWindow:

   ```
   dw_detail.inv_linkage.of_SetMaster(dw_master)
   ```

4. Register the related columns by calling the of_Register function:

   ```
   dw_detail.inv_linkage.of_Register &
          ("emp_id","emp_id")
   ```

5. (Optional) Specify that the service updates DataWindows from the bottom of the linkage chain on up (the default is to update top down):

   ```
   dw_detail.inv_linkage.of_SetUpdateStyle  &

     (dw_detail.inv_linkage.BOTTOMUP)
   ```

6. Establish the action taken by the detail when a row changes in the master by calling the of_SetStyle function.

   This example specifies that the detail DataWindow retrieves a row whenever the master changes:

   ```
   dw_detail.inv_linkage.of_SetStyle  &
     (dw_detail.inv_linkage.RETRIEVE)
   ```

7. Call the master DataWindow’s of_Retrieve function:

   ```
   IF dw_master.of_Retrieve( ) = -1 THEN
         MessageBox("Error","Retrieve error")
   ```

```
ELSE
      dw_master.SetFocus( )
END IF
```

Previous steps can be in one script
You can code the previous steps in a single event, such as the window Open event.

8 Add retrieval logic to the master DataWindow's pfc_Retrieve event:

```
Return this.Retrieve( )
```

Retrieving rows
If the linkage service refreshes detail rows via retrieval, you only need to code a Retrieve function for the master DataWindow. With the filter and scroll options, you must also code Retrieve functions in detail DataWindows.

v **To enable confirm on row change (retrieval style only):**

1 Call the of_SetUpdateOnRowChange function for the detail DataWindow:

```
dw_detail.inv_linkage.of_SetUpdateOnRowChange(TRUE)
```

2 Call the of_SetConfirmOnRowChange function for the detail DataWindow:

```
dw_detail.inv_linkage.of_SetConfirmOnRowChange  &
   (TRUE)
```

v **To enable confirm on delete:**

1 Call the of_SetUpdateOnRowChange function for the detail DataWindow:

```
dw_detail.inv_linkage.of_SetUpdateOnRowChange(TRUE)
```

2 Call the of_SetConfirmOnDelete function for the detail DataWindow:

```
dw_detail.inv_linkage.of_SetConfirmOnDelete(TRUE)
```

v **To enable cascading key changes:**

- Call the of_SetSyncOnKeyChange function for every DataWindow in the linkage chain:

```
dw_master.inv_linkage.of_SetSyncOnKeyChange(TRUE)

dw_detail.inv_linkage.of_SetSyncOnKeyChange(TRUE)
```

v **To specify deletion style:**

- Call the of_SetDeleteStyle function for all master DataWindows in the linkage chain:

```
dw_master.inv_linkage.of_SetDeleteStyle  &

   (dw_cust.inv_linkage.DISCARD_ROWS)
```

v **To enable extended update:**

- Call the of_SetOtherSaveObjects function to add other controls to the default save process:

```
PowerObject  lpo_objs[ ]

// This example adds the lv_salesinfo ListView
// to the save process.
lpo_objs[1] = lv_salesinfo
dw_master.inv_linkage.of_SetOtherSaveObjects  &
  (lpo_objs)
```

Multitable update service

Overview

The PFC multitable update service makes it easy for you to update DataWindows containing columns from multiple tables.

PFC enables multitable update services through the n_cst_dwsrv_multitable user object.

DataStore services
This service is available to the n_ds DataStore via the n_cst_dssrv_multitable user object.

Usage

Use this service when you need to update rows for a DataWindow that contains data from more than one table. When you call the w_master pfc_Save event, PFC updates all specified tables in all DataWindows on the window.

v **To enable the multitable update service:**

- Call the u_dw of_SetMultiTable function:

```
dw_emplist.of_SetMultiTable(TRUE)
```

  U_dw destroys the service automatically when the DataWindow is destroyed.

v **To specify the tables to be updated:**

- Call the of_Register function once for each table to be updated in a multitable update:

```
String  ls_projcols[ ] =  &
       {"proj_id"}
String  ls_taskcols[ ] =  &
       {"proj_id", "task_id"}

dw_project.inv_multitable.of_Register  &
       ("project", ls_projcols)
dw_project.inv_multitable.of_Register  &
       ("task", ls_taskcols)
```

v **(Optional) To update a DataWindow that contains data from multiple database tables:**

- Call the w_master pfc_Save event:

```
Integer  li_return

li_return = w_sheet.Event pfc_Save()
...
```

Print preview service

Overview

The PFC print preview service enables you to provide DataWindow print preview capabilities:

- Print preview
- First page, next page, previous page, last page
- Zoom

Menus that descend from PFC's m_master menu have automatic access to this functionality.

PFC enables the print preview service through the n_cst_dwsrv_printpreview user object.

DataStore services
This service is available to the n_ds DataStore via the n_cst_dssrv_printpreview user object.

Usage

Use this service to provide print preview capabilities in your applications. Users enter print preview mode by selecting File>Print Preview from the menu bar.

v **To enable the print preview service:**

- Call the u_dw of_SetPrintPreview function:

```
dw_emplist.of_SetPrintPreview(TRUE)
```

  U_dw destroys the service automatically when the DataWindow is destroyed.

DataWindow properties service

Overview

The DataWindow properties service enables display of the DataWindow Properties window, which allows you to:

- Selectively enable and disable DataWindow services
- View the PFC syntax for the selected service
- Access and modify DataWindow properties interactively, including:

  DataWindow buffers
  Row and column status
  Statistics
  Properties of all objects on the DataWindow object

See "DataWindow Properties window" on page 207.

Usage

Use this service to enable display of the DataWindow Properties window.

v **To enable the DataWindow properties service:**

1 Call the u_dw of_SetProperty function:

```
dw_emplist.of_SetProperty(TRUE)
```

U_dw destroys the service automatically when the DataWindow is destroyed.

2 When the DataWindow displays, right-click and select DataWindow Properties.

The DataWindow Properties window displays.

Query mode service

Overview

The PFC query mode service makes it easier for you to provide query mode capabilities in applications. The service also helps users to understand and use query mode.

While in query mode, users can right-click to display a pop-up menu with options that display columns, operators, and values.

For complete information on DataWindow query mode, see the *PowerBuilder User's Guide*.

PFC enables the query mode service through the n_cst_dwsrv_querymode user object.

Usage

Use this service for the following:

- Beginning and ending query mode
- Specifying the columns eligible for query mode
- Saving queries to a file and loading previously saved queries

v **To enable the query mode service:**

- Call the u_dw of_SetQuerymode function:

```
dw_emplist.of_SetQuerymode(TRUE)
```

U_dw destroys the service automatically when the DataWindow is destroyed.

v **To begin query mode:**

- Call the of_SetEnabled function, passing TRUE:

```
dw_emplist.inv_querymode.of_SetEnabled(TRUE)
```

v **To end query mode:**

- Call the of_SetEnabled function, passing FALSE:

```
dw_emplist.inv_querymode.of_SetEnabled(FALSE)
```

v **To specify columns eligible for query mode:**

- Call the of_SetQueryCols function, passing an array listing the columns eligible for query mode:

```
String  ls_cols[]

ls_cols[1] = "emp_dept_id"
ls_cols[2] = "emp_id"
dw_emplist.inv_querymode.of_SetQueryCols(ls_cols)
```

  When you call of_SetEnabled, the query mode service protects ineligible columns.

v **To save a query to a file:**

1. Start query mode by calling of_SetEnabled (TRUE).
2. Allow the user to specify query mode criteria.
3. Call the of_Save function.

   This function displays a dialog box that prompts the user for the name of the file in which to save the query.

v **To load a query from a file:**

- Call the of_Load function.

  This function displays a dialog box that prompts the user to select a saved query from disk. If the user selects a file, this function uses the selected file to determine selection criteria.

Reporting service

Overview

The PFC reporting service allows you to provide enhanced viewing and printing capabilities in an application's DataWindows.

Many of this service's functions provide the option of either executing the DataWindow Modify function or returning Modify syntax for use as input to your own Modify function. If you code more than two consecutive report service functions, consider returning the Modify syntax, concatenating the strings and issuing the Modify function from within your own code.

DataWindows must use PBUs or pixels
To use this service, the DataWindow object must use PBUs or pixels as the DataWindow Unit. It does not work with DataWindows that use thousandths of an inch or thousandths of a centimeter as the DataWindow Unit.

PFC enables reporting services through the n_cst_dwsrv_report user object.

DataStore services
This service is available to the n_ds DataStore via the n_cst_dssrv_report user object.

Usage

Use this service for the following:

- Adding items to a DataWindow
- Creating a composite DataWindow out of one or more individual DataWindows (allowing multiple DataWindows to print as a single report)
- Formatting and printing
- Setting background, color, and border
- Zooming a DataWindow relative to its current size

v **To enable the reporting service:**

- Call the u_dw of_SetReport function:

```
dw_emplist.of_SetReport(TRUE)
```

U_dw destroys the service automatically when the DataWindow is destroyed.

v **To add items to a DataWindow:**

- Call one of the following n_cst_dwsrv_report functions:

Function	What it does
of_AddCompute	Adds a computed column
of_AddLine	Adds a line
of_AddPicture	Adds a bitmap
of_AddText	Adds text

v **To create a composite DataWindow:**

1. Call the of_CreateComposite function, passing information on the DataWindows to be included in the composite:

```
String   ls_dws[ ], ls_trailfooter[ ]
String   ls_slide[ ]

String   ls_return
Integer  li_return
Boolean  lb_vertical
Border   lbo_border[ ]

lb_vertical = TRUE
ls_dws[1] = "d_employee"
ls_dws[2] = "d_benefits"
ls_trailfooter[1] = "No"
ls_trailfooter[2] = "Yes"
ls_slide[1] = "AllAbove"
ls_slide[2] = "AllAbove"
lbo_border[1] = Lowered!
lbo_border[2] = Lowered!

li_Return = &
       dw_composite.inv_report.of_CreateComposite &
       (ls_dws, lb_vertical, ls_trailfooter, &
       ls_slide, lbo_border)
IF li_Return = 1 THEN
       dw_composite.SetTransObject(SQLCA)
       dw_composite.Event pfc_Retrieve( )
END IF
```

2. Print or display the composite DataWindow as appropriate.

```
dw_composite.inv_report.of_PrintReport &
       (TRUE, FALSE)
```

v **To print a DataWindow:**

- Call the of_PrintReport function.

v **To set defaults, color, and border:**

- Call one of the following n_cst_dwsrv_report functions:

Function	What it does
of_SetDefaultBackColor of_SetDefaultBorder of_SetDefaultCharset of_SetDefaultColor of_SetDefaultFontFace of_SetDefaultFontSize	Modifies DataWindow defaults
of_SetBorder	Modifies the border for one or more objects in a DataWindow
of_SetColor	Modifies the color and background color (if applicable) of one or more objects in a DataWindow

v **To control DataWindow zoom:**

- Call the of_SetRelativeZoom function.

Zoom is relative to the current display
Of_SetRelativeZoom modifies zoom percentage relative to the current zoom percentage. For example, if a DataWindow is currently displayed at 80% and you specify of_SetRelativeZoom (50), the function changes the zoom percentage to 40%.

Required column service

Overview

The PFC required column service. This service enables and disables default DataWindow processing for required fields. This makes it easier for your application to handle interdependent fields within a DataWindow.

This service applies only to DataWindow columns that use the nilisnull property. For example, EditMasks don’t have this property, so the required column service doesn’t apply to edit masks.

PFC enables the required column service through the n_cst_dwsrv_reqcolumn user object.

Usage

DataWindow required fields processing can interfere with the user-directed interface offered by a GUI application. The required column service allows you to defer required fields processing until the user completes data entry.

The service allows you to specify columns for which PowerBuilder should still perform required fields processing.

Required fields checking
When you call the window's pfc_Save event, it automatically performs required fields checking before updating the database.

v **To enable the required column service:**

- Call the u_dw of_SetReqColumn function:

  ```
  dw_emplist.of_SetReqColumn(TRUE)
  ```

  U_dw destroys the service automatically when the DataWindow is destroyed.

v **To override the service for certain columns:**

- Call the of_RegisterSkipColumn to specify which columns should retain standard PowerBuilder required fields processing:

  ```
  dw_emplist.inv_reqcolumn.of_RegisterSkipColumn  &
          ("dept_id")
  ```

Row management service

Overview

The PFC row management service allows you to insert and delete rows. The row management service also provides a function to undo row deletions.

PFC enables the row management service through the n_cst_dwsrv_rowmanager user object.

Usage

Use this service for the following:

- Adding an empty row to the end of the DataWindow
- Inserting an empty row between two existing rows
- Deleting one or more rows
- Displaying a dialog box allowing you to restore deleted rows

v **To enable the row management service:**

- Call the u_dw of_SetRowManager function:

```
dw_emplist.of_SetRowManager(TRUE)
```

U_dw destroys the service automatically when the DataWindow is destroyed.

v **To add an empty row to the end of the DataWindow:**

- Call the pfc_AddRow event:

```
Long  ll_return

ll_return =  &
      dw_emplist.inv_rowmanager.Event pfc_AddRow()
IF ll_return = -1 THEN
      MessageBox("Error", "Error adding empty row")
END IF
```

PFC calls this event automatically when the user selects Add from the m_dw pop-up menu.

v **To insert an empty row between two existing rows:**

- Call the pfc_InsertRow event.

This example inserts before the current row:

```
Long  ll_return

ll_return = dw_emplist.inv_rowmanager.Event &
      pfc_InsertRow()
IF ll_return = -1 THEN
      MessageBox("Error", "Insert error")
END IF
```

PFC calls this event automatically when the user selects Insert from the m_dw pop-up menu.

v **To delete rows:**

- Call the pfc_DeleteRow event.

This example deletes the current row or all selected rows:

```
Long  ll_return

ll_return =  &
      dw_emplist.inv_rowmanager.pfc_DeleteRow()
```

```
IF ll_return = -1 THEN
      MessageBox("Error", "Deletion error")
END IF
```

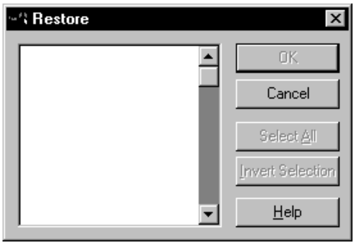
PFC calls this event automatically when the user selects Delete from the m_dw pop-up menu.

To allow users to select multiple rows, use the row selection service.

v **To restore deleted rows:**

- Call the pfc_RestoreRow event.

  This event calls the of_UnDelete function, which displays the w_restorerow dialog box, allowing users to restore deleted rows:



Row selection service

Overview

The PFC row selection service allows you to provide single-, multi-, and extended selection capabilities in a DataWindow.

PFC enables the row selection service through the n_cst_dwsrv_rowselection user object.

Usage

The row selection service handles all row selection automatically. All you have to do is enable the service and specify the desired selection style:

- **Single-row selection** Handles row selection when your DataWindow allows one row to be selected at a time.
- **Multirow selection** Handles row selection by allowing your runtime users to select multiple rows with single clicks. These rows can be contiguous or noncontiguous.

  When multirow selection is enabled, runtime users toggle a row's selected state by clicking it. This capability is similar to a list box's MultiSelect attribute.

- **Extended selection** Handles row selection by allowing your runtime users to select multiple rows with SHIFT+click and CTRL+click.

  When extended selection is enabled, runtime users select multiple contiguous rows using SHIFT+click and noncontiguous rows using CTRL+click. This capability is similar to a list box's ExtendedSelect attribute.

v **To enable the row selection service:**

- Call the u_dw of_SetRowSelect function:

```
dw_emplist.of_SetRowSelect(TRUE)
```

  U_dw destroys the service automatically when the DataWindow is destroyed.

v **To specify the row selection style:**

- Call the of_SetStyle function, passing the selection style you want. This example enables extended selection:

```
dw_emplist.inv_rowselect.of_SetStyle  &
      (dw_emplist.inv_rowselect.EXTENDED)
```

DataWindow resize service

Overview

The DataWindow resize service allows you to resize the columns in a DataWindow control when the user resizes the window.

Use this service to add resize capabilities to the columns that display in a DataWindow.

PFC enables the DataWindow resize service through the n_cst_dwsrv_resize user object.

Usage

You use the DataWindow resize service to enable resizing of the objects displayed in a DataWindow object (so that when the user resizes the window, this service resizes DataWindow contents automatically).

Presentation styles
You cannot use the DataWindow resize service with DataWindow objects that have the Composite or RichTextEdit presentation style.

This service provides two resizing options:

- **For simple resizing** Call the of_Register function passing n_cst_dwsrv_resize constants, such as FIXEDBOTTOM
- **For total control over resizing** Implement weighted resize by calling the of_Register function with explicit specifications for moving and scaling

v **To enable the DataWindow resize service:**

1 Call the u_dw of_SetResize function, set the Transaction object, and specify that Sort dialog boxes use DataWindow column header names:

```
dw_emp.of_SetResize(TRUE)
```

U_dw destroys the service automatically when the DataWindow is destroyed.

2 (Optional) Specify the DataWindow control's original size by calling the of_SetOrigSize function. You call this function if an MDI application opens MDI sheets with an enumeration other than Original!:

```
this.inv_resize.of_SetOrigSize  &
       (this.width, this.height)
```

3 (Optional) Call the of_SetMinSize function to specify a minimum size below which the DataWindow resize service no longer resizes DataWindow contents:

```
this.inv_resize.of_SetMinSize  &
       (this.width-50, this.height-100)
```

4 Specify the columns to be resized and how they should be resized by calling the of_Register function:

```
this.inv_resize.of_Register("emp_fname",  &
       0, 0, 50, 50)
this.inv_resize.of_Register("emp_lname",  &
       100, 0, 50, 50)
```

5 Enable the window resize service and register the DataWindow control (this example is from a window Open event):

```
this.of_SetResize(TRUE)
this.inv_resize.of_Register(dw_1, 0, 0, 100, 100)
```

6 (Optional) Call the of_UnRegister function to remove columns from the resize list.

Sort service

Overview

The PFC sort service allows you to provide easy-to-use sort capabilities in a DataWindow.

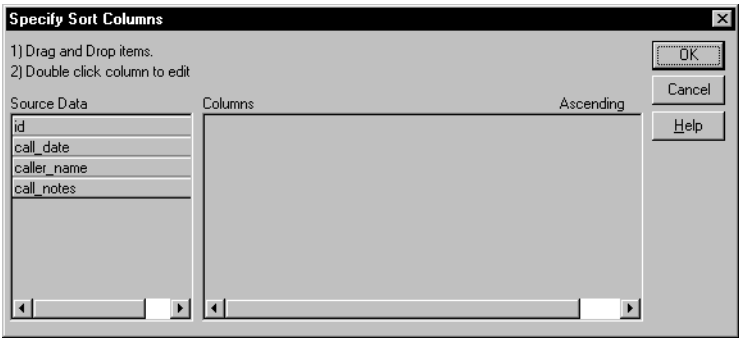
Use this service to add sort capabilities to your application. For example, you might add a menu item that calls the pfc_SortDlg event.

PFC enables the sort service through the n_cst_dwsrv_sort user object.

Usage

The sort service displays Sort dialog boxes automatically. All you do is enable the service and specify the sort style you want. You can choose among four styles of sort dialog boxes:

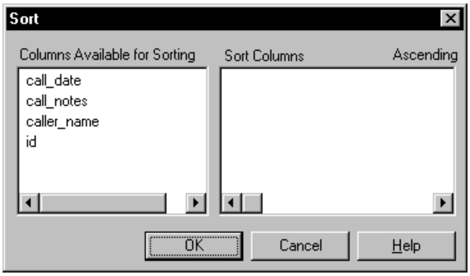
- Default PowerBuilder Sort dialog box:



1) Drag and Drop items.
2) Double click column to edit

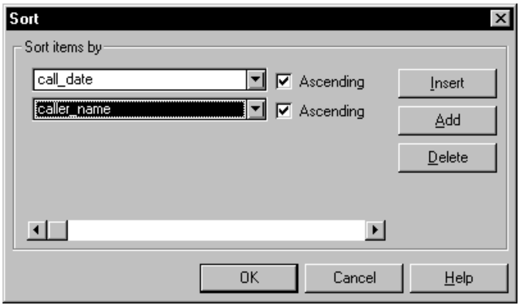
id
call_date
caller_name
call_notes

- Drag and drop sorting (w_sortdragdrop dialog box):



call_date
call_notes
caller_name
id

- Multicolumn sorting (w_sortmulti dialog box):



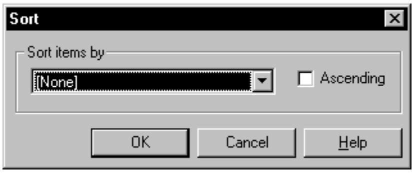
Sort

Sort items by

call_date ☑ Ascending

caller_name ☑ Ascending

- Single-column sort (pfc_w_sortsingle dialog box):



Sort

Sort items by

[None] ☐ Ascending

Additionally, you can allow the user to sort by clicking on column headings (for column header sorting, the column header object must be in the primary header band of the DataWindow)

v **To enable the sort service:**

- Call the u_dw of_SetSort function and specify that Sort dialog boxes use DataWindow column header names:

```
dw_emp.of_SetSort(TRUE)
dw_emp.inv_sort.of_SetColumnDisplayNameStyle  &
   (dw_emp.inv_sort.HEADER)
```

Sorting by column header
If you sort by column header, make sure that all columns added to the DataWindow have headers, and that these conform to the naming scheme for headers. The default naming scheme uses the suffix _t, but you can change this by calling the of_SetDefaultHeaderSuffix function.

U_dw destroys the service automatically when the DataWindow is destroyed.

v **To specify whether PFC sort dialog boxes sort on display values or data values:**

- Call the of_SetUseDisplay function:

```
dw_emp.inv_sort.of_SetUseDisplay(TRUE)
```

v **To specify the sort style:**

- Call the of_SetStyle function, specifying the Sort dialog box type:

```
dw_emp.inv_sort.of_SetStyle  &
   (dw_emp.inv_sort.DRAGDROP)
```

v **To display the Sort dialog box:**

- Call the u_dw pfc_SortDlg event:

```
dw_emplist.Event pfc_SortDlg( )
```

You do not typically call this event. In most cases, the user displays the Sort dialog box by selecting View>Sort from the menu bar of a menu that descends from PFC's m_master menu.

Window services

PFC implements window services through:

- Functions, events, and instance variables coded in w_master and its descendants
- Custom class user objects

To access window services, you create windows that descend from one of PFC's w_master descendants:

w_child
w_frame
w_main
w_popup
w_response
w_sheet

W_master contains:

- Functions to enable and disable window services implemented as custom class user objects

- Instance variables that allow you to reference each custom class user object’s functions, events, and instance variables (this type of instance variable is called a reference variable)
- Precoded events and user events that perform window services and call custom class user object functions
- Empty user events to which you add code to perform application-specific processing

Inherit from windows in the extension level
When using windows, always inherit from windows with the w_ prefix (don’t inherit from windows with the pfc_ prefix). Pfc_ prefixed objects are subject to change when you upgrade PFC versions.

The following table lists window services and how they are implemented:

Window service	Implementation
Basic window service	Implemented in n_cst_winsrv and as well as functions and user events in PFC windows
Preference service	n_cst_winsrv_preference
Sheet management service	n_cst_winsrv_sheetmanager
Status bar service	n_cst_winsrv_statusbar

Basic window services

Overview

PFC windows include:

- Window functions
- Precoded events and user events
- Empty user events

These functions and events are available to all of your application’s windows. PFC implements much of this functionality automatically when you use PFC windows in conjunction with PFC visual user objects and menus that descend from PFC’s m_master menu.

Automatic CloseQuery processing
PFC window services include automatic CloseQuery processing for all DataWindows in a window. This processing saves all pending changes if the user clicks Yes in the Save Changes dialog box.

If you want to implement application-specific save processing, override the CloseQuery event in your application's windows. (To do this globally, override CloseQuery in w_master or disable CloseQuery processing by setting the w_master ib_disableclosequery instance variable to TRUE.)

Usage

Basic PFC window functionality includes:

- Message router and menu integration
- Empty user events, which are triggered by PFC menu items
- Toolbar control (w_frame only)
- Automatic save processing, implemented through the logical unit of work service

For information on using a specific PFC window type, see the window's discussion in the *PFC Object Reference*.

v **To use the message router from a menu item script:**

- Call the of_SendMessage menu function, passing the user event to be called:

  ```
  of_SendMessage("pfc_CheckStatus")
  ```

  The of_SendMessage function passes the request to n_cst_menu of_SendMessage function, which calls the current window's pfc_MessageRouter user event, which calls the specified user event automatically.

v **To use the message router from within a nonmenu function or event:**

- Call the active window's pfc_MessageRouter user event, passing the user event to be called:

  ```
  this.Event pfc_MessageRouter("pfc_CheckStatus")
  ```

  The pfc_MessageRouter event passes the request to the current window, which triggers the specified user event automatically.

PFC menus use the message router
PFC menus use the of_SendMessage menu function to call PFC user events on a window.

v **To use empty user events:**

- Add code to the PFC user event that performs the intended processing. This example, which you might code in the pfc_PageSetup user event, displays a PageSetup dialog box for the current DataWindow:

```
Integer  li_return

li_return = idw_active.Event pfc_PageSetup()
IF li_return > 0 THEN
      li_return = idw_active.Event &
         pfc_PrintImmediate()
END IF
```

  The discussions in the *PFC Object Reference* show which events require additional coding.

v **To display a dialog box that allows users to control toolbars:**

- Call the frame window’s pfc_Toolbars user event:

```
gnv_app.of_GetFrame().Event pfc_Toolbars()
```

  This dialog box displays automatically when the user selects Tools>Customize Toolbars from a menu that descends from PFC’s m_master menu.

v **To save changes to the database:**

- Call the window’s pfc_Save user event:

```
Integer  li_return

li_return = this.Event pfc_Save()
```

  PFC menus call this user event when the user selects File>Save from a menu that descends from PFC’s m_master menu. Additionally, the w_master CloseQuery event calls pfc_Save if the user clicks Yes when prompted to save changes.

v **To center a window on the screen:**

1 Enable the base window service:

```
this.of_SetBase(TRUE)
```

2 Call the n_cst_winsrv of_Center function:

```
this.inv_base.of_Center()
```

Preference service

Overview

The PFC preference service provides functionality that automatically saves and restores a user's window settings using either the registry or an INI file. The preference services saves:

- Size
- Position
- Toolbar settings

PFC enables the preference service through the n_cst_winsrv_preference user object.

Usage

Use this service to save and restore window settings.

Automatic resetting
If you enable the preference service, windows descended from w_master save and restore settings automatically.

v **To enable the window preference service:**

- Call the w_master of_SetPreference function. This function is available in all windows developed with PFC (w_master is the ancestor of all PFC windows):

```
this.of_SetPreference(TRUE)
```

PFC destroys the service automatically when the window closes.

v **To specify which window settings to restore, call one or more of the following functions:**

- Call the following functions as needed:

  of_SetToolbarItemOrder
  of_SetToolbarItemSpace
  of_SetToolbarItemVisible
  of_SetToolbars
  of_SetToolbarTitles
  of_SetWindow

Sheet management service

Overview

The PFC sheet management service provides functions that help you manage multiple sheets in an MDI application. When you enable the sheet management service, PFC enables these items on the Window menu:

- Minimize All Windows
- Undo Arrange Icons

PFC enables the sheet management service through the n_cst_winsrv_sheetmanager user object.

Usage

Use this service to manage multiple sheets in MDI applications.

v **To enable the window sheet management service:**

- Call the w_frame of_SetSheetManager function. This function is available in all windows that descend from w_frame:

```
this.of_SetSheetManager(TRUE)
```

PFC destroys the service automatically when the frame window closes.

v **To access sheet information:**

- Call the following functions as needed:

  of_GetSheetCount
  of_GetSheets
  of_GetSheetsByClass
  of_GetSheetsByTitle

PFC destroys the service automatically when the frame window closes.

Status bar service

Overview

The PFC status bar service displays date, time, and memory information in the lower-right corner of an MDI frame window. Other status bar service features include:

- Threshold monitoring for GDI and free memory
- Progress bar support
- Display of developer-specific text

For information on progress bar display, see “Using the progress bar control” on page 183.

PFC enables the status bar service through the n_cst_winsrv_statusbar user object. Status bar information displays in the w_statusbar pop-up window.

You call n_cst_winsrv_statusbar functions to control the items displayed.

Usage

Use this service to display status bar information in an MDI frame window.

If necessary, you can call w_statusbar functions to modify status bar information via PowerScript code.

v **To enable the window status bar service:**

1 Call the w_frame of_SetStatusBar function. This function is available in all windows that descend from w_frame:

```
this.of_SetStatusBar(TRUE)
```

PFC destroys the service automatically when the frame window closes.

2 Call n_cst_winsrv_statusbar functions in the w_frame pfc_PreOpen event to specify the items displayed. The service displays items in the order that their associated functions are called, from left to right. For example, the following example displays memory to the left of the date and time:

```
this.inv_statusbar.of_SetMem(TRUE)
this.inv_statusbar.of_SetTimer(TRUE)
```

3 Call other n_cst_winsrv_statusbar function as appropriate.

Menu service

Overview

The PFC menu service provides functions that help you communicate between a menu and a window. It also provides functions that return information on an MDI frame and toolbar items. You use the menu service functions within menu item scripts.

PFC enables the menu service through the n_cst_menu user object.

Usage

Use this service in non-PFC menus to access the frame window and to communicate with windows.

v **To enable the menu service:**

- Declare a variable of type n_cst_menu:

```
n_cst_menu  lnv_menu
```

This can be a menu-level instance variable or a local variable within each menu item script.

Because PFC defines this object with the autoinstantiate option, you don’t need to code CREATE or DESTROY statements.

v **To use the message router:**

- Call the of_SendMessage function from within a menu item script:

```
n_cst_menu  lnv_menu

Message.StringParm = "w_emplist"
lnv_menu.of_SendMessage(this, "pfc_Open")
```

v **To access the frame window:**

- Call the of_GetMDIFrame function from within a menu item script (this example accesses the MDI frame to use in calling a frame-level event):

```
n_cst_menu  lnv_menu
w_frame  lw_frame

// This is an alternative to of_SendMessage.
lnv_menu.of_GetMDIFrame(this, lw_frame)
Message.StringParm = "w_emplist"
lw_frame.Event pfc_Open()
```

Resize service

Overview

The PFC resize service provides functions that automatically move and resize controls when the user resizes a window, tab, or tab page. This service allows you to control how and whether controls resize when the window, tab, or tab page resizes.

PFC enables the resize service through the n_cst_resize user object.

You use n_cst_dwsrv_resize, the DataWindow resize service to move and resize columns within a DataWindow.

Usage

Use this service to control window resizing. It provides two resizing options:

- **For simple resizing** Call the of_Register function passing n_cst_resize constants, such as FIXEDBOTTOM
- **For total control over resizing** Implement weighted resize by calling the of_Register function with explicit specifications for moving and scaling

v **To enable the resize service:**

- Call the w_master, u_tab, or u_tabpg of_SetResize function:

  ```
  this.of_SetResize(TRUE)
  ```

  PFC destroys the service automatically when the window or tab closes.

v **To register resizable controls:**

- Call the of_Register function specifying how the control should respond when the window or tab resize. For each registered control, you specify how much the control should move during resize and how much the control should scale during resize. This example lets a DataWindow control expand down and to the right:

  ```
  this.inv_resize.of_Register(dw_emplist,  &
          0, 100, 100, 100)
  ```

v **To specify a minimum size below which the resize service no longer resizes controls:**

- Call the of_SetMinSize function, specifying a minimum size. You might place this code in the window Open event, specifying a minimum size somewhat smaller than the original size:

  ```
  Integer   li_return

  li_return = this.inv_resize.of_SetMinSize  &
          (this.width - 200, this.height - 150)
  ```

v **To use the resize service with sheets in an MDI application:**

- Use either of the following methods:
  - Open sheets in their original size:

    ```
    OpenSheet(w_emp, "w_emplist", w_frame,  &
       0 , Original!)
    ```

- If you open sheets with any other enumeration, call the of_SetOrigSize function before registering controls with the resize service. The call to of_SetOrigSize passes what the width and height would have been had the sheet opened in the original size:

```
this.inv_resize.of_SetOrigSize(1200, 1000)
```

Conversion service

Overview

The PFC conversion service provides functions that you can call to convert values from one data type to another. For example, you can call the of_Boolean function to convert an integer or a string into a boolean value.

PFC enables the conversion service through the n_cst_conversion user object. N_cst_conversion uses the PowerBuilder autoinstantiate option, which eliminates the need for CREATE and DESTROY statements.

Usage

You can use conversion service functions to convert:

From	To
Integer or String	Boolean
Boolean, ToolbarAlignment, or SQLPreviewType	String
Boolean	Integer
String	ToolbarAlignment
Button	String
Icon	String
String	SQLPreviewType

For complete information on conversion service functions, see the discussion about n_cst_conversion in the *PFC Object Reference*.

Defining n_cst_conversion

Define n_cst_conversion as a global, instance, or local variable, as appropriate for your application:

Usage of conversion functions	Variable type
Throughout your application	Global variable or as an instance variable on n_cst_appmanager
Within a single object	Instance variable for the object
Within a single script	Local variable

v **To enable the conversion service:**

- Declare a variable of type n_cst_conversion.

```
n_cst_conversion   inv_conversion
```

Because PFC defines this object with the autoinstantiate option, you don't need to code CREATE or DESTROY statements.

v **To call a conversion service function:**

- Call the function, using dot notation to specify the object instance.

This example assumes an inv_conversion instance variable:

```
String   ls_checked

ls_checked = inv_conversion.of_String  &
       (cbx_confirmed.Enabled)
MessageBox("Conversion", "CheckBox is: "  &
       + ls_checked)
```

Date/Time service

Overview

The PFC date/time service provides functions that you can call to perform calculations with dates. For example, you can call the of_SecondsAfter function to determine the number of seconds between two date/time values.

PFC enables the date/time service through the n_cst_datetime user object. N_cst_datetime uses the PowerBuilder autoinstantiate option, which eliminates the need for CREATE and DESTROY statements.

Usage

You can use the date/time service to perform many date and time calculations. Functions you can perform with the date/time service include:

- Convert a Julian date to a Gregorian date (Gregorian dates use the Date datatype)
- Convert seconds to hours
- Convert seconds to days
- Convert a Gregorian date to a Julian date
- Determine the number of years between two date/time values
- Determine the number of months between two date/time values

- Determine the number of weeks between two date/time values
- Determine the number of seconds between two date/time values
- Determine the number of milliseconds between two date/time values
- Determine if a date is valid
- Determine if a date falls on a weekday
- Determine if a date falls on a weekend
- Halt processing until a specified date/time

Define n_cst_datetime as a global, instance, or local variable, as appropriate for your application.

Usage of date/time functions	Variable type
Throughout your application	Global variable or as an instance variable on n_cst_appmanager
Within a single object	Instance variable for the object
Within a single script	Local variable

v **To enable the date/time service:**

- Declare a variable of type n_cst_datetime:

```
n_cst_datetime   inv_datetime
```

Because PFC defines this object with the autoinstantiate option, you don’t need to code CREATE or DESTROY statements.

v **To call a date/time service function:**

- Call the function, using dot notation to specify the object instance.

This example assumes an inv_datetime instance variable:

```
Long   ll_seconds, ll_days

ll_seconds = Long(sle_seconds.Text)
ll_days = inv_datetime.of_Days(ll_seconds)

MessageBox("Date/Time", &
  String(ll_seconds) + " seconds is equal to " + &
    String(ll_days) + " days.")
```

File service

Overview

The PFC file service provides functions that you can call to add file-management functionality to an application. For example, you can call the of_FileRename function to rename a file.

The file service includes support for many platform-specific types of operations, automatically calling the appropriate external function.

PFC enables the file service through the n_cst_filesrv user object and its platform-specific descendants.

Usage

Actions you can perform with the file service include:

- Assembling a concatenated filename
- Creating and deleting directories
- Reading, writing, renaming, and copying files, including files larger than 32,765 bytes
- Accessing file information, including date and time
- Creating and sorting a list of all files in a directory

Define n_cst_filesrv as a global, instance, or local variable, as appropriate for your application:

Usage of file service functions	Variable type
Throughout your application	Global variable or as an instance variable on n_cst_appmanager
Within a single object	Instance variable for the object
Within a single script	Local variable

Because PFC instantiates a platform-specific descendant of n_cst_filesrv, it does not use the autoinstantiate feature. You must explicitly destroy the n_cst_filesrv instance when you are through.

v **To enable the file service:**

1. Declare a variable of type n_cst_filesrv:

   ```
   n_cst_filesrv  inv_filesrv
   ```

2. Call the f_set_filesrv global function:

   ```
   f_SetFilesrv(inv_filesrv, TRUE)
   ```

   The f_SetFilesrv global function automatically creates the platform-specific n_cst_filesrv descendant.

3 Destroy the n_cst_filesrv object when you are through:

```
DESTROY inv_filesrv
```

v **To call a file service function:**

- Call the function, using dot notation to specify the object instance.

  This example calls the of_FileRead function to access the contents of the file specified in the sle_filename SingleLineEdit. The example assumes an inv_filesrv instance variable:

```
Integer  li_return
String   ls_file[ ]

li_return = inv_filesrv.of_FileRead  &
   (sle_filename.text, ls_file)
CHOOSE CASE li_return
   CASE -1
      MessageBox("Error", "Error accessing file")
   CASE ELSE
      // File processing goes here
END CHOOSE
```

v **To destroy the file service:**

- Use the DESTROY statement:

```
DESTROY inv_filesrv
```

INI file service

Overview

The PFC INI file service provides functions that you can call to read from and write to INI files.

PFC enables the INI file service through the n_cst_inifile user object.

Usage

You can use the INI file service to:

- Retrieve all keys for an INI-file section
- Retrieve all sections for an INI file
- Remove a line from the INI file
- Remove an entire section from the INI file

Using ProfileInt, ProfileString, and SetProfileString
Use the ProfileInt, ProfileString, and SetProfileString PowerScript functions to access INI file entries one at a time.

The INI file service is not case sensitive.

v **To enable the INI file service:**

- Declare a variable of type n_cst_inifile:

```
n_cst_inifile  inv_ini_handler
```

  Because PFC defines this object with the autoinstantiate option, you don't need to code CREATE or DESTROY statements.

v **To use the INI file service:**

- Call n_cst_inifile object functions as needed, using dot notation to specify the object instance.

  This example, which displays all of an INI file section's keys in a ListBox, assumes an inv_ini_handler instance variable:

```
String   ls_keys[ ]
Integer  li_count, li_size

li_size = inv_ini_handler.of_GetKeys  &
       (gnv_app.of_GetAppINIFile(), "CustApp",
ls_keys)
lb_keys.Reset( )
FOR li_count = 1 to li_size
       lb_keys.AddItem(ls_keys[li_count])
NEXT
```

Numerical service

Overview

The PFC numerical service provides functions that you can call to access binary data. For example, you can call the of_GetBit function to determine if a specified bit is on or off.

PFC enables the numerical service through the n_cst_numerical user object.

Usage

You can use numerical service functions to:

- Determine whether a specified bit is on or off
- Convert a base 10 number to binary
- Convert a binary number to base 10

Use this object with the Windows SDK
The Windows Software Development Kit (SDK) includes many functions that return bit values. Use the of_GetBit function to access these values.

Define n_cst_numerical as a global, instance, or local variable, as appropriate for your application:

Usage of numerical functions	Variable type
Throughout your application	Global variable or as an instance variable on n_cst_appmanager
Within a single object	Instance variable for the object
Within a single script	Local variable

v **To enable the numerical service:**

- Declare a variable of type n_cst_numerical:

```
n_cst_numerical  inv_numerical
```

Because PFC defines this object with the autoinstantiate option, you don’t need to code CREATE or DESTROY statements.

v **To call a numerical service function:**

- Call the function, using dot notation to specify the object instance.

This example assumes an inv_numerical instance variable:

```
Long    ll_base10
String  ls_binary

ll_base10 = Long(sle_base10.text)
ls_binary = inv_numerical.of_Binary(ll_base10)
MessageBox("Numerical",  &
       String(ll_base10) + " base 10 is equal to " &
          + ls_binary + " in binary.")
```

Platform service

Overview

The PFC platform service provides functions that you can call to add platform-specific functionality to an application. You can use this service's functions on multiple platforms without recoding or adding conditional logic that checks for the current platform. For example, you can call the of_GetFreeMemory function to determine the amount of remaining memory; the platform service automatically calls the appropriate external function for the current platform.

PFC enables platform services through the n_cst_platform user object and its platform-specific descendants.

Print and Page Setup dialog boxes
PFC enables the Print and Page Setup dialog boxes through the platform service.

Usage

Functions you can perform with the platform service include:

- Determining the amount of free memory
- Determining the amount of free system resources
- Determining the height and width, in PBUs, given a text string

Define n_cst_platform as a global, instance, or local variable, as appropriate for your application:

Usage of platform functions	Variable type
Throughout your application	Global variable or as an instance variable on n_cst_appmanager
Within a single object	Instance variable for the object
Within a single script	Local variable

Because PFC instantiates a platform-specific descendant of n_cst_platform, it does not use the autoinstantiate feature. You must explicitly destroy the n_cst_platform instance when you are through.

v **To enable the platform service:**

1 Declare a variable of type n_cst_platform:

```
n_cst_platform  inv_platform
```

2 Call the f_SetPlatform global function:

```
f_SetPlatform(inv_platform, TRUE)
```

The f_SetPlatform global function automatically creates the platform-specific n_cst_platform descendant.

v **To call a platform service function:**

- Call the function, using dot notation to specify the object instance.

  This example calls the of_GetFreememory function and displays this value in the status bar. The example assumes an inv_platform instance variable:

```
Long  ll_free_memory

ll_free_memory =  &
      inv_platform.of_GetFreeMemory()
gnv_app.of_GetFrame().SetMicroHelp &
      ("Free memory: " + String(ll_free_memory) )
```

v **To destroy the platform service:**

- Use the DESTROY statement:

```
DESTROY inv_platform
```

Selection service

Overview

The PFC selection service provides a function that displays the w_selection dialog box, which allows users to select a row. When the user clicks OK, the function returns the values in one or more columns for the selected row.

PFC enables the selection service through the n_cst_selection user object.

Usage

You use the selection service's of_Open function to display a dialog box allowing users to choose an item that your application then processes.

There are three basic versions of the of_Open function. Each displays different information in w_selection:

- W_selection retrieves and displays all rows for a specified DataWindow object
- W_selection displays a passed set of rows
- W_selection displays rows that have been saved as part of the passed DataWindow object

Define n_cst_selection as a global, instance, or local variable, as appropriate for your application:

Usage of selection service functions	Variable type
Throughout your application	Global variable or as an instance variable on n_cst_appmanager
Within a single object	Instance variable for the object
Within a single script	Local variable

v **To enable the selection service:**

- Declare a variable of type n_cst_selection:

  ```
  n_cst_selection  inv_selection
  ```

  Because PFC defines this object with the autoinstantiate option, you don't need to code CREATE or DESTROY statements.

v **To use the selection service:**

1. Declare variables used by of_Open:

   ```
   n_cst_selection  lnv_selection
   Any  la_selected[ ]
   String ls_columns[ ]
   Integer  li_count
   ```

2. Specify the columns whose values are to be returned:

   ```
   ls_columns[1] = "emp_id"
   ls_columns[2] = "emp_lname"
   ls_columns[3] = "emp_fname"
   ```

3. Display the w_selection window by calling the of_Open function (this version of of_Open causes w_selection to retrieve all rows in the specified DataWindow object):

   ```
   lnv_selection.of_Open  &
      ("d_empall", la_selected, SQLCA, ls_columns)
   ```

4. Access the returned column values as appropriate (this example displays returned values in a ListBox):

   ```
   FOR li_count = 1 to UpperBound(la_selected)
      lb_selected.AddItem  &
         (String(la_selected[li_count]))
   NEXT
   ```

SQL parsing service

Overview

The PFC SQL parsing service provides functions that you can call to assemble and parse SQL statements.

PFC enables the SQL parsing service through the n_cst_sql user object.

Usage

You can use the SQL parsing service to:

- Build a SQL statement from its component parts
- Parse a SQL statement into its component parts

Define n_cst_sql as a global, instance, or local variable, as appropriate for your application:

Usage of SQL parsing functions	Variable type
Throughout your application	Global variable or as an instance variable on n_cst_appmanager
Within a single object	Instance variable for the object
Within a single script	Local variable

v **To enable the SQL parsing service:**

- Declare a variable of type n_cst_sql:

  ```
  n_cst_sql   inv_sql
  ```

  Because PFC defines this object with the autoinstantiate option, you don’t need to code CREATE or DESTROY statements.

v **To build a SQL statement from its component parts:**

- Call the of_Assemble function, using dot notation to specify the object instance.

  This example assumes an inv_sql instance variable:

  ```
  String  ls_sql
  n_cst_sqlattrib  lnv_sqlattrib[ ]

  lnv_sqlattrib[1].s_verb = sle_verb.text
  lnv_sqlattrib[1].s_tables = sle_tables.text
  lnv_sqlattrib[1].s_columns = sle_columns.text
  lnv_sqlattrib[1].s_values = sle_values.text
  lnv_sqlattrib[1].s_where = sle_where.text
  lnv_sqlattrib[1].s_order = sle_order.text
  lnv_sqlattrib[1].s_group = sle_group.text
  lnv_sqlattrib[1].s_having = sle_having.text
  ```

```
ls_sql = inv_sql.of_Assemble(lstr_sql)
MessageBox("SQL", ls_sql)
```

v **To parse a SQL statement into its component parts:**

- Call the of_Parse function, using dot notation to specify the object instance.

  This example assumes an inv_sql instance variable:

```
String   ls_sql
Integer  li_return
n_cst_sqlattrib  lnv_sqlattrib[ ]

li_return = inv_sql.of_Parse  &
       (mle_sql.text, lnv_sqlattrib)
IF li_return > 0 THEN
      sle_verb.text= lnv_sqlattrib[1].s_verb
      sle_tables.text = lnv_sqlattrib[1].s_tables
      sle_columns.text =
lnv_sqlattrib[1].s_columns
      sle_values.text = lnv_sqlattrib[1].s_values
      sle_where.text = lnv_sqlattrib[1].s_where
      sle_order.text = lnv_sqlattrib[1].s_order
      sle_group.text = lnv_sqlattrib[1].s_group
      sle_having.text= lnv_sqlattrib[1].s_having
END IF
```

String-handling service

Overview

The PFC string-handling service provides functions that you can call to operate on strings.

PFC enables the string-handling service through the n_cst_string user object.

Usage

You can use the string-handling service to perform many string operations, including:

- Separating a delimited string into an array
- Converting an array into a delimited string
- Determining if a string is lowercase, uppercase, alphabetic, or alphanumeric.
- Global replacing

- Counting the number of occurrences of a specified string
- Removing spaces and nonprintable characters from the beginning or end of a string
- Determining if a string is a comparison or arithmetic operator
- Converting all the words in a string to initial cap

Define n_cst_string as a global, instance, or local variable, as appropriate for your application:

Usage of string-handling functions	Variable type
Throughout your application	Global variable or as an instance variable on n_cst_appmanager
Within a single object	Instance variable for the object
Within a single script	Local variable

v **To enable the string-handling service:**

- Declare a variable of type n_cst_string:

```
n_cst_string  inv_string
```

Because PFC defines this object with the autoinstantiate option, you don't need to code CREATE or DESTROY statements.

v **To call a string-handling service function:**

- Call the function, using dot notation to specify the object instance.

This example, which is from the n_cst_dwsrv_report of_AddText function, calls the of_ParseToArray function to convert the as_text string into elements in the ls_line array. The example uses an lnv_string local variable:

```
n_cst_string  lnv_string
Integer  li_newlines
String  ls_line[ ]
...
li_newlines = lnv_string.of_ParseToArray  &
       (as_text, "~r~n", ls_line)
```

Metaclass service

Overview

The metaclass service contains functions that provide information on the functions, events, and variables defined within another object.

PFC enables the string-handling service through the n_cst_metaclass user object.

Usage

The most common use of the metaclass service is to determine whether an object function or event exists before calling it.

v **To use the metaclass service:**

1 Create a variable of type n_cst_metaclass:

```
boolean  lb_defined
n_cst_metaclass  lnv_metaclass
classdefinition  lcd_obj
String ls_args[ ]
Integer  li_rc
```

Because PFC defines this object with the autoinstantiate option, there is no need to code CREATE or DESTROY statements.

2 Call n_cst_metaclass functions as needed:

```
lcd_obj = FindClassDefinition("w_sheet")
lb_defined = lnv_metaclass.of_isFunctionDefined &
  (lcd_obj,"of_Validation", ls_args)
If lb_defined Then
   // Qualify with instance of w_sheet descendant.
   li_rc = w_sheet.Function Dynamic &
     of_Validation ()
       If li_rc < 0 Then Return -1
End If
```

Logical unit of work service

Overview

The logical unit of work service provides support for self-updating objects. A self-updating object encapsulates all required update functionality by implementing a set of functions (self-updating object API) that n_cst_luw calls during the save process. These functions call events that update the object as appropriate. The logical unit of work service calls these functions automatically as part of the default save process.

PFC includes several self-updating objects, including:

U_dw
N_ds
U_lvs
U_tab
U_tvs
U_base
W_master

Examine these objects to see implementations of the self-updating object API.

The default w_master pfc_Save process uses the logical unit of work service to update all updatable self-updating objects on a window.

See "Using the pfc_Save process" on page 195.

Implementing self-updating objects

The logical unit of work service updates all referenced, updatable self-updating objects.

Most self-updating objects are not updatable by default
To ensure backward compatibility, u_dw is the only self-updating object that is updatable by default.

The functions that make up the complete self-updating object API are:

Function	Purpose
of_AcceptText	Calls the pfc_AcceptText event, which calls AcceptText functions as appropriate
of_UpdatesPending	Calls the pfc_UpdatesPending event, which determines whether the object has been updated
of_Validation	Calls the pfc_Validation event, which validates data for the object
of_UpdatePrep	Calls the pfc_UpdatePrep event, which prepares the object for update as appropriate
of_Update	Calls the pfc_Update event, which updates the database
of_PostUpdate	Calls the pfc_PostUpdate event, which performs post-update processing as appropriate

Use the Browser for information on the signatures of these functions and events.

Writing your own self-updating objects

To write a self-updating object, implement the functions listed above and ensure that a reference to the object is in the array passed to the corresponding n_cst_luw functions.

Extending the save process

By default, the w_master pfc_Save process updates all modified DataWindows within the window. You can extend this process as follows:

- **Other self-updating objects** You can define other self-updating objects as updatable by calling the of_SetUpdateable function in the object's Constructor event. This example is from a u_lvs-based ListView:

  ```
  this.of_SetUpdateable(TRUE)
  ```

  Now the logical unit of work service will call functions to update the u_lvs data source as part of a default save process.

- **DataStores** You can add one or more DataStores to the list of objects to be updated by calling the w_master of_SetUpdateObjects function:

  ```
  PowerObject  lpo_objs[ ]
  Integer  li_count

  lpo_objs = this.control
  li_count = UpperBound(lpo_objs)
  li_count++
  lpo_objs[li_count] = ids_data
  this.of_SetUpdateObjects(lpo_objs)
  ```

- **Additional windows** You can add one or more windows to the list of objects to be updated by calling the w_master of_SetUpdateObjects function:

  ```
  PowerObject  lpo_objs[ ]
  Integer  li_count

  lpo_objs = this.control
  li_count = UpperBound(lpo_objs)
  li_count++
  // Update w_other as well as this window
  lpo_objs[li_count] = w_other
  this.of_SetUpdateObjects(lpo_objs)
  ```

List service

Overview

Many applications need to maintain information in linked lists. The PFC list service provides objects and functions you use to create and manipulate linked lists. It supports these types of lists:

- Basic linked list (sorted or unsorted)
- Stack (LIFO)
- Queue (FIFO)
- Tree (balanced binary tree)

A list is made up of nodes

A linked list is made up of **nodes**. Each node contains:

- A reference to the previous item in the list
- A reference to the next item in the list
- A key
- Data
- Balance information (tree lists only)

When adding a node to a linked list, you provide the key and data; the service objects maintain references to the previous and next items.

About sorted lists

A tree list is sorted automatically. You can also use the n_cst_list object to maintain a sorted linked list.

Using a basic list

A basic linked list differs from a stack and a queue in that nodes are not removed as they are accessed.

PFC enables basic list processing through the n_cst_list user object.

Creating a basic list

When you create a basic list, you create and populate instances of n_cst_linkedlistnode and add them to the list.

v **To create a basic list:**

1 Declare an instance variable of type n_cst_list:

```
n_cst_list  inv_list
```

Because PFC defines this object with the autoinstantiate option, you don't need to code CREATE or DESTROY statements.

2 Add nodes to the list. To do this, create an instance of n_cst_linkedlistnode, specify a key and data by calling the n_cst_node of_SetKey and of_SetData functions, then add the node to the list by calling of_Add. This example adds a list item using a SingleLineEdit as the source:

```
n_cst_linkedlistnode  lnv_node
Integer  li_return

lnv_node = CREATE n_cst_linkedlistnode
lnv_node.of_SetKey(sle_1.text)
lnv_node.of_SetData(sle_1.text)
li_return = inv_list.of_Add(lnv_node)
```

Creating a sorted list

The default PFC sorted list object maintains a list of nodes in ascending order, by key value. Duplicates are allowed by default; but you can disallow them if necessary. A sorted list differs from a stack and a queue in that nodes are not removed as they are accessed.

You can customize sort processing by extending the n_cst_linkedlistnodecompare of_Compare function.

See “Creating a comparison object” on page 124.

PFC enables sorted list processing through the n_cst_list user object.

v **To create a sorted list:**

1 Declare an instance variable of type n_cst_list:

```
n_cst_list  inv_sortedlist
```

2 Specify that the list is sorted:

```
inv_sortedlist.of_SetSorted(TRUE)
```

3 (Optional) Specify whether the list allows duplicate entries (by default, duplicates are allowed):

```
inv_sortedlist.of_SetDuplicatesAllowed(FALSE)
```

4 (Optional) Specify a customized node comparison object. This example assumes an inv_customcompare instance variable of type n_cst_customcompare:

```
inv_customecompare = CREATE n_cst_customcompare
inv_sortedlist.of_SetCompare(n_cst_customcompare)
```

5 Add nodes to the list. To do this, first create an instance of n_cst_linkedlistnode, specify a key and data by calling the n_cst_node of_SetKey and of_SetData functions, then add the node to the list by calling of_Add. This example adds a sorted list item using a SingleLineEdit as the source:

```
n_cst_linkedlistnode  lnv_node
Integer  li_return

lnv_node = CREATE n_cst_linkedlistnode
lnv_node.of_SetKey(sle_1.text)
lnv_node.of_SetData(sle_1.text)
li_return = inv_list.of_Add(lnv_node)
```

Finding nodes in a list

To find nodes you must first know the key. PFC does not remove list nodes as they are accessed.

v **To find a node in a sorted list:**

1 Define variables for two nodes:

```
n_cst_linkedlistnode  lnv_node, lnv_temp
```

2 Create one of the nodes:

```
lnv_temp = CREATE n_cst_linkedlistnode
```

3 Populate the empty node with the key of the node you want to find:

```
lnv_temp.of_SetKey(sle_2.text)
```

4 Call the of_Find function to access the requested node. This function’s first argument returns a reference to the requested node:

```
inv_list.of_Find(lnv_node, lnv_temp)
```

5 Access the key and data of the found node by calling the n_cst_linkedlistnode of_GetKey and of_GetData functions.

6 Destroy the temporary node:

```
DESTROY lnv_temp
```

Do not destroy lnv_node

The lnv_node variable is a reference to the node in the list. If you destroy it, the list becomes corrupted. Instead, use of_Remove as described in the next procedure.

Accessing the entire list at once

You can use the of_Get function to retrieve all nodes in the list.

v **To retrieve the entire list in one function call:**

1 Define an array of type n_cst_linkedlistnode to contain the retrieved nodes:

```
n_cst_linkedlistnode  lnv_nodes[ ]
Integer  li_return, li_count
Any  la_data
String  ls_data
```

2 Call the of_Get function:

```
li_return = inv_list.of_Get(lnv_nodes)
```

3 Process the nodes as appropriate. This example displays node data in a ListBox:

```
lb_1.Reset()
FOR li_count = 1 to li_return
      lnv_nodes[li_count].of_GetData(la_data)
      ls_data = String(la_data)
      lb_1.AddItem(ls_data)
NEXT
```

This technique also applies to stacks, queues, and trees.

Removing nodes from a list

You must explicitly remove nodes from a list (they are not removed automatically, as in a stack or a queue).

v **To remove a node from a list:**

1 Define variables for two nodes:

```
n_cst_linkedlistnode  lnv_node, lnv_temp
```

2 Create one of the nodes:

```
lnv_temp = CREATE n_cst_linkedlistnode
```

3 Populate the empty node with the key of the node you want to remove:

```
lnv_temp.of_SetKey(sle_2.text)
```

4 Call the of_Find function to access the requested node. This function returns a reference to the requested node (the first argument):

```
inv_list.of_Find(lnv_node, lnv_temp)
```

5 Remove the node from the list by calling the of_Remove function:

```
inv_list.of_Remove(lnv_node)
```

6 Destroy the temporary node:

```
DESTROY lnv_temp
```

Destroying a list

When you no longer need the list, destroy the list and all of its nodes.

v **To destroy a list and all of its nodes:**

- Destroy all nodes in the list by calling of_Destroy:

```
Long  ll_count

ll_count = inv_list.of_Destroy()
MessageBox("Destroy List",  &
      String(ll_count) + " nodes destroyed")
```

Using a stack

A stack maintains a last-in first-out (LIFO) list. When you add a new node to the stack, the stack object places it at the beginning; when you get a node from the stack, the stack object accesses it from the beginning of the list, removing it in the process.

PFC enables stack processing through the n_cst_stack user object.

Creating a stack

When you create a stack, you create and populate instances of n_cst_linkedlistnode and push them onto the stack.

v **To create a stack:**

1 Declare an instance variable of type n_cst_stack:

```
n_cst_stack   inv_stack
```

Because PFC defines this object with the autoinstantiate option, you don’t need to code CREATE or DESTROY statements.

2 Add nodes to the stack. To do this, create an instance of the node, specify a key and data by calling the n_cst_linkedlistnode of_SetKey and of_SetData functions, then add the node to the stack by calling of_Push (this example creates a stack using a SingleLineEdit as the source):

```
n_cst_linkedlistnode  lnv_node
Integer  li_return
```

```
lnv_node = CREATE n_cst_linkedlistnode
lnv_node.of_SetKey(sle_1.text)
lnv_node.of_SetData(sle_1.text)
li_return = inv_stack.of_Push(lnv_node)
```

Removing nodes from a stack

You can only access a node from the beginning of the stack (the last node added). You pop a stack to access a node.

v **To remove a node from the stack:**

1 Declare a variable of type n_cst_linkedlistnode to contain the node you want to remove from the stack:

```
n_cst_linkedlistnode  lnv_node
```

2 Remove the node from the stack by calling of_Pop:

```
inv_stack.of_Pop(lnv_node)
```

3 Access the key and data as necessary by calling the n_cst_linkedlistnode of_GetKey and of_GetData functions:

```
Any  la_key

IF IsValid(lnv_node) THEN
      lnv_node.of_GetKey(la_key)
      MessageBox("Stack",  &
     "Key is " + String(la_key))
ELSE
      MessageBox("Stack", "List is empty")
END IF
```

Destroying a stack

When you no longer need the stack, destroy the stack and all of its nodes.

v **To destroy a stack:**

- Destroy all nodes in the stack by calling of_Destroy:

```
Long  ll_count

ll_count = inv_stack.of_Destroy()
MessageBox("Destroy",  &
      String(ll_count) + " nodes destroyed")
```

Using a queue

A queue maintains a first-in, first-out (FIFO) list. When you add a new node to the queue, the queue object places it at the end; when you get a node from the queue, the queue object accesses it from the beginning of the list, removing it in the process.

PFC enables queue processing through the n_cst_queue user object.

Creating a queue

When you create a queue, you create and populate instances of n_cst_linkedlistnode and add them to the queue.

v **To create a queue:**

1 Declare an instance variable of type n_cst_queue:

```
n_cst_queue   inv_queue
```

Because PFC defines this object with the autoinstantiate option, you don’t need to code CREATE or DESTROY statements.

2 Add nodes to the queue. To do this, create an instance of the node, specify a key and data by calling the n_cst_linkedlistnode of_SetKey and of_SetData functions, then add the node to the queue by calling of_Put. This example creates a queue using a SingleLineEdit as the source:

```
n_cst_linkedlistnode  lnv_node
Integer  li_return

lnv_node = CREATE n_cst_linkedlistnode
lnv_node.of_SetKey(sle_1.text)
lnv_node.of_SetData(sle_1.text)
li_return = inv_queue.of_Put(lnv_node)
```

Removing nodes from a queue

You can only access nodes from the beginning of a queue (the oldest node in the list).

v **To remove a node from the queue:**

1 Declare a variable of type n_cst_linkedlistnode to contain the node you want to remove from the queue:

```
n_cst_linkedlistnode  lnv_node
```

2 Remove the node from the queue by calling of_Get:

```
inv_queue.of_Get(lnv_node)
```

3 Access the key and data as necessary by calling the n_cst_node of_GetKey and of_GetData functions:

```
Any  la_key

IF IsValid(lnv_node) THEN
lnv_node.of_GetKey(la_key)
MessageBox("Queue",  &
      "Key is " + String(la_key))
ELSE
MessageBox("Queue", "List is empty")
END IF
```

Destroying a queue

When you no longer need the queue, destroy the list and all of its nodes.

v **To destroy a queue:**

- Destroy all nodes in the queue by calling of_Destroy:

```
Long  ll_count

ll_count = inv_queue.of_Destroy()
MessageBox("Destroy All",  &
       String(ll_count) + " nodes destroyed")
```

Using a tree

The PFC tree list object maintains a balanced binary tree of nodes in ascending order, by key value. This object provides all the functionality of a sorted list with no duplicates allowed. It differs from a stack and a queue in that nodes are not removed as they are accessed.

The balanced binary tree maintained by the PFC tree list object is never more than one level out of balance. Because the tree structure reduces the number of nodes that must be searched during find operations, it provides better performance than a sorted list.

You can customize sort processing by extending the n_cst_treenodecompare of_Compare function. See “Creating a comparison object” on page 124.

PFC enables tree processing through the n_cst_tree user object.

Creating a tree

When you create a tree, you create and populate instances of n_cst_treenode and add them to the tree.

v **To create a tree:**

1 Declare an instance variable of type n_cst_tree:

```
n_cst_tree  inv_tree
```

Because PFC defines this object with the autoinstantiate option, you don’t need to code CREATE or DESTROY statements.

2 (Optional) Specify a customized node comparison object. This example assumes an inv_customcompare instance variable of type n_cst_customcompare:

```
inv_customecompare = CREATE n_cst_customcompare
inv_tree.of_SetCompare(inv_customcompare)
```

3 Add nodes to the tree. To do this, create an instance of the node by calling of_Create, specify a key and data by calling the n_cst_treenode of_SetKey and of_SetData functions, then add the node to the tree by calling the of_Add function. This example creates a tree using a SingleLineEdit as the source:

```
n_cst_treenode  lnv_node
Integer  li_return

inv_tree.of_Create(lnv_node)
lnv_node.of_SetKey(sle_1.text)
lnv_node.of_SetData(sle_1.text)
li_return = inv_tree.of_Add(lnv_node)
```

Finding nodes in a tree

You find nodes in a balanced binary tree. They aren’t removed from the list as they are accessed.

v **To find a node in a tree list:**

1 Create an empty node:

```
n_cst_treenode  lnv_node, lnv_temp

inv_tree.of_Create(lnv_temp)
```

2 Populate the empty node with the key of the node you want to find:

```
lnv_temp.of_SetKey(li_key)
```

3 Call the of_Find function to access the requested node. This function’s first argument returns a reference to the requested node:

```
inv_tree.of_Find(lnv_node, lnv_temp)
```

4 Access the key and data of the found node by calling the n_cst_node of_GetKey and of_GetData functions.

5 Destroy the temporary node:

```
DESTROY lnv_temp
```

Do not destroy lnv_node
The lnv_node variable is a reference to the node in the tree list. If you destroy it, the list becomes corrupted. Instead, use of_Remove as described in the next procedure.

Removing nodes from a tree

You must explicitly remove nodes from a tree list (they are not removed automatically, as in a stack or a queue).

v **To remove a node from a tree list:**

1 Create an empty node:

```
n_cst_treenode  lnv_node, lnv_temp

inv_tree.of_Create(lnv_temp)
```

2 Populate the empty node with the key of the node you want to delete:

```
lnv_temp.of_SetKey(li_key)
```

3 Call the of_Find function to access the requested node. This function returns a reference to the requested node (the first argument):

```
inv_tree.of_Find(lnv_node, lnv_temp)
```

4 Remove the node from the tree list by calling the of_Remove function:

```
inv_tree.of_Remove(lnv_node)
```

5 Destroy the temporary node:

```
DESTROY lnv_temp
```

Destroying a tree

When you no longer need the tree list, destroy the list and all of its nodes.

v **To destroy a tree list:**

- Destroy all nodes in the tree list by calling of_Destroy:

```
Long  ll_count

ll_count = inv_tree.of_Destroy()
MessageBox("Destroy All",  &
      String(ll_count) + " nodes destroyed")
```

Creating a comparison object

At the core of any sort processing is a greater than/less than comparison. The PFC sorted list and tree list objects use the n_cst_nodecompare of_Compare function to perform this comparison. By default, the n_cst_nodecompare of_Compare function performs a comparison of two nodes as follows:

- Compares key values (not data values)
- Works for simple data types only (that is, all but object instances and enumerated data types)
- Returns values that the sorted list and tree list objects use to maintain an ascending sorted list

If your sorted list requires different comparison logic, you must inherit from n_cst_nodecompare and override the of_Compare function.

Custom comparison objects

If your sorted list requires different comparison logic, you need to create a descendant of n_cst_nodecompare with an overridden of_Compare function and enable that object at execution time.

v **To create a customized comparison object:**

1 Use the User Object painter to create a customized n_cst_nodecompare descendant.

2 In the customized n_cst_nodecompare descendant, implement a Public of_Compare function to compare key values in the two passed nodes. This function should take two arguments of type n_cst_node (passed by value) and return Integer values as follows:

- **1** The key of the second node is greater than the key of the first node
- **0** The key of the second node is equal to the key of the first node
- **-1** The key of the second node is less than the key of the first node

In this example, each passed node contains a reference to a custom class user object with state and last name instance variables to compare:

```
Any  la_key1, la_key2
String  ls_keytype1, ls_keytype2
n_cst_empinfo  lnv_emp1, lnv_emp2

IF NOT IsValid(anv_node1) THEN Return -3
IF NOT IsValid(anv_node2) THEN Return -3

anv_node1.of_GetKey(la_key1)
IF IsNull(la_key1) THEN Return -4
```

```
anv_node2.of_GetKey(la_key2)
IF IsNull(la_key2) THEN Return -4
ls_keytype1 = ClassName(la_key1)
ls_keytype2 = ClassName(la_key2)

// Check data type of node data.
IF ls_keytype1 = "" THEN Return -6
IF IsNull(ls_keytype1) THEN Return -6
IF ls_keytype1 <> "n_cst_empinfo" THEN Return -6
IF ls_keytype2 = "" THEN Return -6
IF IsNull(ls_keytype2) THEN Return -6
IF ls_keytype2 <> "n_cst_empinfo" THEN Return -6

lnv_emp1 = la_key1 // Cast to n_cst_empinfo
lnv_emp2 = la_key2
// First compare State.
// Additional error checking omitted.
IF lnv_emp1.is_state < lnv_emp2.is_state THEN
       Return -1
ELSEIF lnv_emp1.is_state > lnv_emp2.is_state THEN
       Return 1
ELSE   // States are equal.  Compare last name.
       IF lnv_emp1.is_lname < lnv_emp2.is_lname THEN
          Return -1
       ELSEIF  &
      lnv_emp1.is_lname > lnv_emp2.is_lname THEN
         Return 1
       ELSE  // State and lname are equal.
          Return 0
       END IF
END IF
```

v **To enable a customized comparison object at execution time:**

1 In the object that uses PFC list processing, define an instance variable that uses your customized n_cst_nodecompare object as the data type:

```
n_cst_customcompare   inv_customcompare
```

2 Create an instance of the customized comparison object and call the n_cst_list of_SetCompare function:

```
inv_customcompare = CREATE n_cst_customcompare

inv_sortedlist.of_SetCompare(inv_customcompare)
```

3 Initialize objects before adding them to the list. In the example ahead, you create an array of n_cst_empinfo objects and initialize them with last name, first name, and state.

4 Create nodes, set node values, and add them to the list as necessary:

```
n_cst_node  lnv_node
Integer   li_i

FOR li_i = 1 TO UpperBound(inv_empinfo)
 inv_tree.of_Create(lnv_node)
 lnv_node.of_SetKey(inv_empinfo[li_i])
 lnv_node.of_SetData(inv_empinfo[li_i])
 inv_tree.of_Add(lnv_node)
NEXT
```

Timing service

Overview

The timing service works with PFC's n_tmg Timing object to provide single and multiple timers. These timers are especially useful with standard class and custom class user objects.

PFC enables the timing service through n_tmg, n_cst_tmgsingle, and n_cst_tmgmultiple.

Using single timers

Use n_cst_tmgsingle to maintain a single timer. You establish the timer by calling the of_Register function, specifying the object to be notified, the event to be notified, and the timer interval.

v **To use a single timer:**

1 Establish an instance variable of type n_tmg:

```
n_tmg  itmg_timer
```

2 Create the instance of n_tmg:

```
itmg_timer = CREATE n_tmg
```

3 Enable the single timer service:

```
itmg_timer.of_SetSingle(TRUE)
```

4 Register the object and event to be notified (the object is a window in this example):

```
itmg_timer.inv_single.of_Register  &
   (this, "ue_showtimer", 15)
```

5 Code the event to receive notification from n_cst_tmgsingle (ue_showtimer in this example).

Using multiple timers

Use n_cst_tmgmultiple to maintain multiple timers. You establish each timer by calling the of_Register function, specifying the object to be notified, the event to be notified, and the timer interval.

v **To use multiple timers:**

1 Establish an instance variable of type n_tmg:

```
n_tmg  itmg_timer
```

2 Create the instance of n_tmg:

```
itmg_timer = CREATE n_tmg
```

3 Enable the multiple timer service:

```
itmg_timer.of_SetMultiple(TRUE)
```

4 Register the objects and events to be notified:

```
itmg_timer.inv_multiple.of_Register  &
   (iw_sheet1, "ue_timer", 7)
itmg_timer.inv_multiple.of_Register  &
   (iw_sheet2, "ue_timer", 11)
itmg_timer.inv_multiple.of_Register  &
   (iw_sheet3, "ue_timer", 13)
```

5 Code the events to receive notification from n_cst_tmgmultiple (ue_timer in this example).

CHAPTER 5 Using PFC Visual Controls

About this chapter

This chapter explains how to use PFC standard visual user objects and custom visual user objects.

Contents

About PFC visual controls

PFC contains two types of visual controls:

- **Standard visual user objects** Consist of a single PowerBuilder control. PFC adds logic to enhance the control's functionality and reusability. The u_lb ListBox control is an example of a standard visual user object.
- **Custom visual user objects** Consist of several controls that function as a unit. PFC adds logic to perform the appropriate processing. The u_calculator calculator control is an example of a custom visual object.

Standard class user objects
PFC also features a set of standard *class* user objects, such as n_tr (transaction), n_cn (connection), and n_msg (message).

For information on using standard class user objects, see Chapter 3, "PFC Programming Basics".

Using standard visual user objects

PFC contains standard visual user objects for all window controls. Standard visual user objects include:

- Basic window controls (such as CommandButton, RadioButton, and CheckBox)
- More complex window controls (such as DataWindow, ListView, TreeView, RichTextEdit, and Tab)

Using basic functionality

A standard visual user object inherits its definition from one standard PowerBuilder control. PFC extends each control as appropriate, and you can extend them further.

The PFC standard visual user objects include the following basic functionality, depending on their type:

- **Cut, copy, and paste** Editable controls include Cut, Copy, Paste, and other editing functions.
- **Pop-up menu** Editable controls include code in the RButtonUp event to display a pop-up menu. This menu enables users to Cut, Copy, and Paste text into the current visual control.
- **Autoscroll** The DropDownListBox and DropDownPictureListBox controls provide functionality that scrolls the list automatically as the user types.
- **Selection inversion** The ListBox and PictureListBox controls provide functionality that invert the current selection.
- **Autoselect** Certain editable controls provide functionality that select text automatically when the control receives focus.
- **MicroHelp display** Most controls contain precoded functionality in the GetFocus event to display a tag value in the microhelp area of an MDI frame

For how to place a standard visual user object on a window, see the *PowerBuilder User's Guide*.

Cut, copy, paste, and other editing functions

All editable PFC standard visual user objects include user events that perform text editing functions. Editable PFC visual user objects include:

Control	PFC visual user object
DropDownListBox	u_ddlb
DropDownPictureListBox	u_ddplb
DataWindow	u_dw
EditMask	u_em
MultiLineEdit	u_mle
OLE custom control	u_oc
RichTextEdit	u_rte
SingleLineEdit	u_sle

PFC implements text editing functions by defining user events as appropriate for editable standard visual user objects:

Text editing function	User event
Clear	pfc_Clear
Copy	pfc_Copy
Cut	pfc_Cut
Paste	pfc_Paste
Select All	pfc_SelectAll
Undo	pfc_Undo
Paste Special	pfc_PasteSpecial (u_oc only)

When you use the PFC standard visual user objects as window controls, editing functions are enabled automatically when you use a menu that descends from the PFC m_master menu. M_master includes an Edit menu that has menu items for all editing functions. These Edit menu items use the message router to call the appropriate user event in the current control. You can also add code to command buttons and other controls to call these user events.

Using right-mouse button support

All editable PFC standard visual user objects include right mouse button support, displaying one of the following pop-up menus:

Menu	Used by
m_edit	U_ddlb U_ddplb U_em U_mle U_rte U_sle
m_dw	U_dw
m_lvs	U_lvs
m_oc	U_oc
m_tvs	U_tvs

These pop-up menus all include standard text editing functions. However, text editing functions are not enabled or visible in all pop-up menus by default.

Customizing menu display

All controls that provide right-mouse button support include a pfc_PreRMBMenu event for customizing the items that appear on a pop-up menu. PFC calls this event after the menu is created but before it is displayed.

v **To customize pop-up menu display:**

1. (Optional) Use the Menu painter to create additional items for the pop-up menu.
2. After placing the user object in a window, add logic to the pfc_PreRMBMenu event to hide or disable menu items. This example disables the m_dw Insert, Add Row, and Delete menu items (am_dw is an argument passed by reference to pfc_PreRMBMenu):

```
am_dw.m_table.m_insert.Enabled = FALSE
am_dw.m_table.m_addrow.Enabled = FALSE
am_dw.m_table.m_delete.Enabled = FALSE
```

Disabling right-mouse button support

You can disable right-mouse button support entirely. You may want to do this for a read-only control, for example.

v **To disable right-mouse button support for editable controls:**

- After placing the user object in a window, add code to the control's Constructor event to set the ib_rmbmenu instance variable to FALSE:

```
this.ib_rmbmenu = FALSE
```

U_dw disables items automatically
The u_dw DataWindow control disables pop-up menu items automatically for read-only DataWindow objects.

Using autoscroll in drop-down lists

The u_ddlb DropDownListBox and u_ddplb DropDownPictureListBox controls feature autoscrolling: if you press r, the control scrolls to the first entry beginning with r, selecting the remaining text; if you then press i, the control scrolls to the first entry beginning with ri.

This capability differs from the standard DropDownListBox and DropDownPictureListBox behavior, which scrolls based on the first letter only: if you press r, the control scrolls to the first entry beginning with r; if you then press i, the control scrolls to the first entry beginning with i.

For information on autoscroll in DropDownDataWindows, see “Drop-down DataWindow search service” on page 67.

By default, autoscroll is disabled.

v **To enable autoscroll:**

- After placing the user object in a window, add code to the Constructor event to set the ib_search instance variable to TRUE:

```
this.ib_search = TRUE
```

Using autoselect

These editable controls feature automatic selection:

U_ddlb
U_ddplb
U_em
U_mle
U_sle

When autoselect is enabled, PFC automatically selects all text in the control when it receives focus.

By default, autoselect is disabled.

v **To enable autoselect:**

- After placing the user object in a window, add code to the Constructor event to set the ib_autoselect instance variable to TRUE:

```
this.ib_autoselect = TRUE
```

Using selection inversion in list boxes

The u_lb ListBox and the u_plb PictureListBox controls feature selection inversion. When you call the control's pfc_InvertSelect event, PFC highlights previously unhighlighted items and unhighlights previously highlighted items. Selection inversion is a feature found in many Windows 95 applications.

Extended select or multiselect
To use this feature, you must enable either the Extended Select property or the Multi Select property for the control.

v **To enable selection inversion, use either of the following techniques:**

1 Add a menu item that uses the message router to call the pfc_InvertSelection event. This example adds an Invert Selection menu item that calls the of_SendMessage function to trigger the pfc_InvertSelection event (this method requires that the ListBox or PictureListBox have focus when the user selects the menu item):

```
of_SendMessage("pfc_InvertSelection")
```

2 Add a window control that calls the pfc_InvertSelection event directly (this example is from a CommandButton Clicked event):

```
lb_choices.Event pfc_InvertSelection()
```

Using the GetFocus event

PFC standard visual controls include logic in the GetFocus event to provide focus-related functionality. This event calls the window's pfc_ControlGotFocus user event, which (when MicroHelp display is enabled) updates MicroHelp for controls placed on descendants of w_sheet.

The w_sheet window extends the pfc_ControlGotFocus user event to add automatic MicroHelp display. This feature displays text from the control's tag value in the MDI frame's status bar.

v **To update MicroHelp automatically in a sheet window:**

1 Call the n_cst_appmanager of_SetMicroHelp function to enable MicroHelp display:

```
gnv_app.of_SetMicroHelp(TRUE)
```

2 Define tag values for each of the sheet's controls. Use the following format:

```
MicroHelp=tagtext
```

PFC uses the specified tag text to update MicroHelp automatically when the control gets focus. If there are multiple items in the tag, separate them with semicolons.

Using advanced functionality

To get the most out of PFC, you need to program using advanced controls for which PFC provides additional capabilities:

Control	PFC visual user object
DataWindow	U_dw
ListView	U_lvs
TreeView	U_tvs
RichTextEdit	U_rte
OleControl	U_oc
Tab	U_tab
Tab page	U_tabpg

Using the u_dw DataWindow control

Most production-strength PowerBuilder applications make intense use of DataWindow controls. The u_dw DataWindow control contains extensive built-in methods including:

- Functions to enable and disable DataWindow services
- A function to set the Transaction object
- Events to retrieve rows for DataWindows and DropDownDataWindows
- Events to control DataWindow update
- Events to control printing

Integration with PFC menus
Many of the events described in this section are called automatically by menus that descend from the PFC m_master menu. For example, when you select File>Save from the menu bar, PFC calls the pfc_Save event.

Enabling DataWindow services

PFC provides a variety of DataWindow services that you can use to add production-strength features to an application. Many of these services require little or no coding on your part.

v **To use DataWindow services:**

1 Place the u_dw DataWindow visual user object on the window.

2 Determine which DataWindow services are appropriate for the DataWindow object displayed in the u_dw-DataWindow control.

3 Enable the appropriate DataWindow services, using the u_dw of_Set*servicename* functions (this example from the DataWindow control's Constructor event enables the row selection, row management, and sort services):

```
this.of_SetRowSelect(TRUE)
this.of_SetRowManager(TRUE)
this.of_SetSort(TRUE)
```

4 Establish the Transaction object for the DataWindow:

```
this.of_SetTransObject(SQLCA)
```

5 Call other functions as necessary to initialize services (this example sets the row selection style, specifies the Sort dialog box style, and enables column header sorting):

```
this.inv_rowselect.of_SetStyle &
  (this.inv_rowselect.EXTENDED)
this.inv_sort.of_SetStyle &
  (this.inv_sort.DRAGDROP)
this.inv_sort.of_SetColumnHeader(TRUE)
```

6 Call DataWindow service events and functions as necessary in your application's functions and events. In many cases you don't have to code anything to realize the service's benefits. This example calls the pfc_SortDlg event to display the Sort dialog box:

```
dw_list.Event pfc_SortDlg()
```

Disabling services
The u_dw Destructor event destroys enabled services automatically. In most cases you don't destroy a service explicitly.

For specific usage information on individual DataWindow services, see "DataWindow services" on page 63.

Setting the Transaction object

As shown in the preceding example, you establish a DataWindow's Transaction object by calling the u_dw of_SetTransObject function.

The of_SetTransObject function ensures that the passed Transaction object is valid, sets the Transaction object, and saves a reference to the Transaction object in the itr_object instance variable.

The Transaction object must be of type n_tr.

When using the linkage service
For DataWindows that use the linkage service, call the n_cst_dwsrv_linkage of_SetTransObject function on the top-level DataWindow after all DataWindows have been created and you have established the linkage chain.

For more information on the linkage service, see "Linkage service" on page 71.

Retrieving rows

Because many DataWindow services rely on the u_dw pfc_Retrieve event to retrieve data, it's best to code the PowerScript Retrieve function in the u_dw pfc_Retrieve event. To retrieve rows, your code then calls the u_dw of_Retrieve function, which calls either the pfc_Retrieve event or the n_cst_dwsrv_linkage of_Retrieve function as appropriate.

v **To retrieve rows for a DataWindow:**

1 Call the of_Retrieve function (this example is from a DataWindow Constructor event):

```
Long  ll_return

ll_return = this.of_Retrieve()
```

2 Add code to the pfc_Retrieve event that calls the PowerScript Retrieve function, returning the return code:

```
Return this.Retrieve()
```

Retrieving rows with the linkage service
When using the linkage service to *retrieve* detail DataWindow rows, code the pfc_Retrieve function for the top-level DataWindow only. When using the linkage service to *filter or scroll* detail DataWindow rows, code the pfc_Retrieve event for all DataWindows in the linkage chain.

v **To retrieve rows in a DropDownDataWindow:**

1 Add code to the DataWindow control's pfc_PopulateDDDW event. This code should retrieve rows for the specified DropDownDataWindow:

```
IF as_colname = "dept_id" THEN
  adwc_obj.SetTransObject(SQLCA)
  Return adwc_obj.Retrieve()
ELSE
  Return 0
END IF
```

2 If the DataWindow control is using no other DataWindow services, enable n_cst_dwsrv the base DataWindow service (this example is from the DataWindow control's Constructor event):

```
this.of_SetBase(TRUE)
this.of_SetTransObject(SQLCA)
this.of_Retrieve()
```

3 Call the n_cst_dwsrv of_PopulateDDDWs or of_PopulateDDDW function to update all DropDownDataWindows or a specified DropDownDataWindow:

```
dw_1.inv_base.of_PopulateDDDWs()
// Alternatively, you could call:
// dw_1.inv_base.of_PopulateDDDW("dept_id")
```

Controlling DataWindow updates

Basic DataWindow updates PFC provides two ways to update DataWindows:

- **U_dw pfc_Update event** Updates a single DataWindow without any logical unit of work service processing, automatically calling the n_cst_dwsrv_multitable of_Update function if the multitable update service is enabled
- **W_master pfc_Save event** Uses the logical unit of work service to call the u_dw of_Updatefunction for all DataWindows on the window. For non-PFC DataWindow controls, the logical unit of work service calls the PowerScript Update function

W_master is the ancestor of all PFC windows
Because w_master is the ancestor of all PFC windows, the pfc_Save event is available to all windows in your application.

v **To update a single DataWindow:**

- Call the u_dw pfc_Update event:

```
IF dw_emplist.Event pfc_Update &
   (TRUE, TRUE) = 1 THEN
      SQLCA.of_Commit()
ELSE
   SQLCA.of_Rollback()
END IF
```

v **To update all DataWindows on a window:**

- Call the w_master pfc_Save event:

```
Integer li_return

li_return = w_emp.Event pfc_Save()
IF li_return < 0 THEN
  MessageBox("Update Failed", &
    "Update failed. Return code was " &
      + String(li_return))
ELSE
  gnv_app.of_GetFrame().SetMicroHelp &
    ("Update succeeded")
END IF
```

Automatic CloseQuery processing
If any of a window's DataWindows has pending updates and the user closes the window, PFC displays a Save Changes dialog box automatically. If the user chooses to save changes, CloseQuery processing calls the window's pfc_Save event.

For more information on using pfc_Save, see "Using the pfc_Save process" on page 195.

Declaring nonupdatable DataWindows You can declare a DataWindow as nonupdatable, thus removing it from the pfc_Save update sequence and the PFC default CloseQuery processing.

Shared DataWindows
If your window includes DataWindows that share data, only one DataWindow control should be updatable. All others that share data should be nonupdatable.

v **To declare a DataWindow as nonupdatable:**

- Call the u_dw of_SetUpdatable function:

```
dw_emplist.of_SetUpdateable(FALSE)
```

Printing DataWindows

PFC provides events that allow you to print DataWindows. You can:

- Display a Print dialog box, allowing you to choose options before printing

  PFC uses the s_printdlgattrib structure to pass DataWindow properties to the n_cst_platform of_PrintDlg function. You can use the pfc_PrePrintDlg event to further customize the initial contents of the Print dialog box by modifying elements in the s_printdlgattrib structure.

  The elements in the s_printdlgattrib structure reflect selected DataWindow Print properties (such as collate, page numbers, and number of copies).

- Print a DataWindow without displaying the Print dialog box
- Display a Page Setup dialog box that allows you to specify print settings

  PFC uses the s_pagesetupattrib structure to pass DataWindow properties to the n_cst_platform of_PageSetupDlg function. You can use the pfc_PrePageSetupDlg event to further customize the initial contents of the Page Setup dialog box by modifying elements in the s_pagesetupdlgattrib structure.

  The elements in the s_pagesetupattrib structure reflect selected DataWindow Print properties (such as margins, paper size, and orientation).

v **To display the Print dialog box:**

1. (Optional) Add code to the pfc_PrePrintDlg event to modify the information used by the pfc_PrintDlg function (this example provides a default for the number of copies specification):

```
astr_printdlg.l_copies = 1
```

2. Call the pfc_Print event:

```
dw_emp.Event pfc_Print()
```

v **To print a DataWindow without displaying the Print dialog box:**

- Call the pfc_PrintImmediate event:

```
dw_emp.Event pfc_PrintImmediate()
```

Comparing File>Print with the Print button
When you select File>Print from the menu bar, PFC calls the pfc_Print event. When you click the Print toolbar button, PFC calls the pfc_PrintImmediate event.

v **To display the Page Setup dialog box:**

1 (Optional) Add code to the pfc_PrePageSetupDlg event to modify the information used by the pfc_PageSetupDlg function (this example specifies the initial value for the orientation specification):

```
astr_pagesetup.b_portraitorientation = TRUE
```

2 Call the pfc_PageSetup event:

```
dw_emp.Event pfc_PageSetup()
```

Using the u_lvs ListView control

The u_lvs ListView control makes it easy for you to display and update database data in a ListView. U_lvs includes services that you enable to obtain the features you need:

- **Base service** Provides find functionality and other basic services
- **Data source service** Controls the display and update of database data in a ListView. Also controls the bitmaps displayed with ListView data
- **Sort service** Provides column-header sort functionality (report view only)

The u_lv ListView control
PFC also includes u_lv, a non-service-based ListView that includes many of the same features as u_lvs. U_lv was the PFC ListView control in previous releases and is not documented here.

Displaying database data in a ListView

You use the ListView data source service to associate a ListView with a data source (not to be confused with an ODBC data source). A data source can be:

- DataWindow object (using either data retrieved from the database or data stored with the DataWindow object)
- SQL statement
- DataWindow control
- DataStore control

- Rows from an array
- A file

The ListView data source service (implemented through the n_cst_lvsrv_datasource custom class user object) maintains the ListView's data source in a DataStore and uses it to populate the ListView. You can also specify which columns from the DataWindow object should display when the ListView is in Report view.

v **To establish the initial ListView display:**

1 Add a u_lvs user object to the window.

2 Enable the ListView data source service (this example also enables the ListView sort service):

```
this.of_SetDataSource(TRUE)
this.of_SetSort(TRUE)
```

3 Call the n_cst_lvsrv_datasource of_Register function. This example specifies the DataWindow object, Transaction object, and label column (this example is from a ListView Constructor event):

```
this.inv_datasource.of_Register("emp_lname", &
        "d_emplist", SQLCA)
```

The ListView data source service uses the DataWindow caching service to maintain the data.

4 (Optional) Specify whether right mouse button support is enabled:

```
this.of_SetRMBMenu(TRUE)
```

5 (Optional) Specify additional columns to display in Report view (this example displays all columns) and establish picture information:

```
this.inv_datasource.of_RegisterReportColumn()
this.inv_datasource.of_SetPictureColumn("1")
```

6 (Optional) Declare the ListView as eligible for update via the logical unit of work service and the w_master pfc_Save process:

```
this.of_SetUpdateable(TRUE)
```

7 (Optional) Specify whether PFC asks the user to confirm deletions:

```
this.inv_datasource.of_SetConfirmOnDelete(TRUE)
```

8 Retrieve data from the database and add rows to the ListView:

```
this.Event pfc_Populate()
```

9 Extend the pfc_Retrieve event, adding code that calls the of_Retrieve function, which allows you to specify retrieval arguments:

```
Any   la_args[20]

Return this.of_Retrieve(la_args, ads_data)
```

Deleting items from the ListView

You can delete items from the ListView, optionally deleting them from the data source and the database.

v **To remove items from the ListView, allowing them to remain in the u_lvs DataStore and the database:**

1 Enable the ListView's Delete Items property.

2 Call the PowerScript DeleteItem function:

```
Integer  li_index
li_index = lv_list.SelectedIndex()

lv_list.DeleteItem(li_index)
```

Redisplaying deleted rows
To redisplay all rows in the data source, call the of_Reset function followed by the pfc_Populate event.

v **To delete selected items from the ListView and the database:**

1 Enable the ListView's Delete Items property.

2 Call the pfc_Delete event:

```
lv_1.Event pfc_Delete()
```

3 Call the logical unit of work service of_Save function to update the database (this example is from a user event defined for the ListView control):

```
PowerObject  lpo_obj[ ]
n_tr  ltr_obj[ ]

IF NOT IsValid(inv_luw) THEN
   inv_luw = CREATE n_cst_luw
END IF
lpo_obj[1] = this
ltr_obj[1] = SQLCA

// Error processing omitted to save space
inv_luw.of_Save(lpo_obj, ltr_obj)
```

Inserting items into the ListView

You can use a ListView to insert items into the database. But because the ListView control allows updates to the label column only, you cannot add information into all the columns displayed in Report view. To get around this, you need another mechanism (such as a dialog box) to acquire enough information to update the DataWindow, the ListView, and the database.

v **To insert a row into the database using a ListView:**

1. Create a mechanism (such as a dialog box) that collects information needed to add a new row.
2. Use the information you gathered to call the of_InsertItem function. This example assumes you added information to a temporary DataStore, which is used as input to of_InsertItem:

```
lv_dept.of_InsertItem(ids_newrow, 1)
```

Using pictures

ListViews allow you to specify up to three different pictures to display with an item:

- **Default picture** An image that appears with a Listview item
- **State picture** An image that appears to the left of the original image
- **Overlay picture** An image (typically an icon or cursor) that appears on top of a ListView item's original image, indicating a difference between the ListView item and other items

U_lvs allows you to set up the initial picture display using the following:

- **Picture index** Each index entry points to a bitmap file or PowerBuilder system bitmap that the ListView uses for picture display. You define entries in the picture index using the ListView's property sheet. This approach results in all ListView items displaying the same picture
- **DataWindow column** A column specifying row-specific display information. The column can come directly from the database or can be a DataWindow computed column. It can contain either of the following:
  - A string specifying the name of a bitmap file that the ListView uses when displaying the corresponding row
  - An integer specifying the picture index the ListView uses when displaying the corresponding row

  Using a DataWindow column allows you to customize ListView item display.

For more on ListViews, see *Application Techniques*.

v **To use the picture index to specify ListView pictures:**

1. Establish the picture index using the ListView’s property sheet.
2. Enable the ListView data source service:

   ```
   this.of_SetDataSource(TRUE)
   ```

3. Specify the appropriate picture index entries by calling n_cst_lvsrv_datasource functions, passing the picture index:

   ```
   this.inv_datasource.of_SetPictureColumn('1')
   this.inv_datasource.of_SetOverlayPictureColumn('1')
   this.inv_datasource.of_SetStatePictureColumn('1')
   ```

v **To use DataWindow columns to specify ListView pictures:**

1. Establish database or DataWindow columns and populate them with bitmap names (String or Character data type) or picture index specifications (Integer data type).
2. Enable the ListView data source service:

   ```
   this.of_SetDataSource(TRUE)
   ```

3. Specify the appropriate pictures by calling n_cst_lvsrv_datasource functions, passing the DataWindow column names:

   ```
   this.inv_datasource.of_SetPictureColumn  &
      ('picture_name')
   this.inv_datasource.of_SetOverlayPictureColumn  &
      ('picture_overlay')
   this.inv_datasource.of_SetStatePictureColumn  &
      ('picture_state')
   ```

Bitmaps must exist
The files named in the retrieved rows must exist in a directory accessible to the application.

Using the u_tvs TreeView control

The u_tvs TreeView control makes it easy for you to use DataWindows to display and update hierarchical database data in a TreeView. U_tvs includes services that you enable to obtain the features you need:

- **Base service** Provides basic services
- **Level source service** Controls the display and update of database data in a TreeView level. Also controls the bitmaps displayed with TreeView data

---

The u_tv TreeView control
PFC also includes u_tv, a non-service-based TreeView that includes many of the same features as u_tvs. U_tv was the PFC TreeView control in previous releases and is not documented here.

Basic use

You use the TreeView level source service to associate each TreeView level with a data source (not to be confused with an ODBC data source). A data source can be:

- DataWindow object (using either data retrieved from the database or data stored with the DataWindow object)
- SQL statement
- DataWindow control
- DataStore control
- Rows from an array
- A file

The TreeView level source service (implemented through the n_cst_tvsrv_levelsource custom class user object) maintains each TreeView level's data source in a DataStore and uses it to populate the TreeView level.

Establishing the level's data source You establish a level's data source by calling the n_cst_tvsrv_levelsource of_Register function. This function includes an argument that specifies how a level relates to the levels above it. This argument must be in the format :*scope*.*level*.*column* where:

- *Scope* specifies one of the following literals:
    - Level
    - Parent

- *Level* specifies an absolute or relative level number, depending on what you specify for *scope*:

Scope specification	Level specification	Example
Level	The value you specify indicates an absolute level number	`:level.1.emp_name` indicates that the retrieval argument is from the emp_name column of the item's level-1 ancestor
Parent	The value you specify indicates a level relative to the current level	`:parent.2.emp_name` indicates that the retrieval argument comes from the emp_name column of the ancestor two levels above

- *Column* specifies the DataWindow object column name from which to obtain the values used in retrieval arguments.

For example, the following string specifies that the retrieval argument is from the emp_name column of the item's level-1 ancestor:

```
:level.1.emp_name
```

Multiple retrieval arguments
If a DataWindow object has multiple retrieval arguments, you specify the *scope.level.column* argument multiple times within the same string. For example, the following string specifies that the retrieval arguments are from the region column two levels higher and the states_state_id column one level higher:

```
":parent.2.region, :parent.1.states_state_id"
```

v **To display a TreeView:**

1 Add a u_tvs user object to the window.

2 Enable the level source service using the u_tvs of_SetLevelSource function:

```
this.of_SetLevelSource(TRUE)
```

3 Define a data source for each TreeView level by calling the of_Register function, once for each level:

```
this.inv_levelsource.of_Register(1,  &
   "dept_name", "", "d_deptlist", SQLCA, "")
```

```
this.inv_levelsource.of_Register(2, "emp_lname",  &
  ":parent.1.dept_id", "d_empbydept", SQLCA, "")
```

4 Call additional functions as necessary to control TreeView behavior:

```
this.inv_levelsource.of_SetPictureColumn(1, "1")
this.inv_levelsource.of_SetSelectedPictureColumn  &
   (1, "2")
this.inv_levelsource.of_SetPictureColumn(2, "4")
this.inv_levelsource.of_SetSelectedPictureColumn  &
   (2, "5")
```

5 (Optional) Specify whether right mouse button support is enabled:

```
this.of_SetRMBMenu(TRUE)
```

6 (Optional) Declare the TreeView as eligible for update via the logical unit of work service and the w_master pfc_Save process:

7 Populate the TreeView by calling the u_tvs pfc_Populate event:

```
this.event pfc_Populate(0)
```

8 Extend the pfc_Retrieve event, adding code that calls the of_Retrieve function, specifying retrieval arguments as returned by the n_cst_tvsrv_levelsource of_GetArgs function:

```
Any   la_args[20]
Integer   li_level

IF IsValid(inv_levelsource) THEN
      li_level = this.of_GetNextLevel(al_parent)
      this.inv_levelsource.of_GetArgs(al_parent, &
     li_level, la_args)
END IF

Return this.of_Retrieve(al_parent, la_args,  &
   ads_data)
```

Deleting items from a TreeView

You can delete items from the TreeView, optionally deleting them from the data source and the database.

v **To remove items from the TreeView (allowing them to remain in the data source and the database):**

1 Enable the TreeView's Delete Items property.

2 Call the PowerScript DeleteItem function:

```
Long  ll_tvi

ll_tvi = this.FindItem(CurrentTreeItem!, 0)
```

Redisplaying deleted rows
To redisplay these rows, call the u_tvs pfc_RefreshLevel event.

v **To delete items from the TreeView and the database:**

1. Enable the TreeView's Delete Items property.
2. Call the pfc_Delete event:

```
tv_1.Event pfc_Delete()
```

3. Call the logical unit of work service of_Save function to update the database (this example is from a user event defined for the TreeView control):

```
PowerObject  lpo_obj[ ]
n_tr  ltr_obj[ ]

IF NOT IsValid(inv_luw) THEN
   inv_luw = CREATE n_cst_luw
END IF
lpo_obj[1] = this
ltr_obj[1] = SQLCA

// Error processing omitted to save space
inv_luw.of_Save(lpo_obj, ltr_obj)
```

Inserting items into a TreeView

You can use a TreeView to insert new items into the database. But because the TreeView control displays a single field only, you typically need another mechanism (such as a dialog box) to acquire enough information to update the DataWindow, the TreeView, and the database.

v **To insert a row into the database using a TreeView:**

1. Create a mechanism (such as a dialog box) that collects information needed to add a new row.
2. Use the information you gathered to call the of_InsertItem function. This example assumes you added information to a temporary DataStore, which is used as input to of_InsertItem:

```
Long  ll_handle
TreeViewItem  ltvi_item
Long  ll_return

ll_handle = tv_1.FindItem(CurrentTreeItem! , 0)
ll_return = tv_1.of_InsertItem  &
   (ll_handle, ids_data, 1, "Sorted", 0)
```

Using recursion to populate a TreeView

The u_tvs TreeView control allows you to display multiple levels from a single table that has a recursive relationship. In the employee table, for example, there might be a recursive relationship between managers and employees, with each employee row containing a column that points to its manager's employee ID.

You indicate a recursive relationship through an argument to the n_cst_tvsrv_levelsource of_Register function. A recursive level is always the lowest level specified.

v **To use recursion to populate a TreeView:**

1 Create DataWindow objects to display high-level information as well as the recursive data. In the table that follows, the d_empmanagerrecursive DataWindow object handles all levels of manager below the department head using recursive data:

DataWindow object	Contents	Pseudo WHERE clause
d_dept	Departments	None
d_empdeptmanager	Department heads	employee.emp_id = department.dept_head_id
d_empmanagerrecursive	Managers and their employees	employee.manager_id = manager.emp_id

2 Create a window that has a TreeView based on u_tvs.

3 Enable the level source service:

```
this.of_SetLevelSource(TRUE)
```

4 Call the of_Register function to establish the hierarchy and recursive levels. Then call the pfc_Populate function to retrieve data (this example is from the TreeView's Constructor event):

```
this.inv_levelsource.of_Register(1, "dept_name", &
   "", "d_dept", SQLCA, "")
this.inv_levelsource.of_Register(2,  &
   "dept_head_id", ":Level.1.dept_head_id",  &
      "d_empdeptmanager", SQLCA, "")
this.inv_levelsource.of_Register(3, "emp_lname", &
    ":Parent.1.emp_id", "d_empmanagerrecursive", &
       SQLCA, "")
```

5 Call the n_cst_tvsrv_levelsource of_SetRecursive function for the bottom level:

```
this.inv_levelsource.of_SetRecursive(3, TRUE)
```

6 Code other processing as necessary.

Using pictures

A TreeView consists of items that are associated with one or more pictures, which are used in different ways:

Picture	Usage
Default	Represents a TreeView item in its normal mode
Selected	Represents a selected TreeView item
State	Appears to the left of the TreeView item indicating that the item is not in its normal mode, for example changed or unavailable
Overlay	Appears on top of a TreeView item

For more on how you specify pictures, see “Using pictures” on page 144.

v **To use the picture index to specify TreeView pictures:**

1 Using either the Window painter or the User Object painter, specify default pictures for the TreeView. PFC uses this picture index for default, selected, and overlay pictures.

2 Using either the Window painter or the User Object painter, specify state pictures for the TreeView.

3 Enable the level source service:

```
this.of_SetLevelSource(TRUE)
```

4 Associate pictures with TreeView levels by calling n_cst_tvsrv_levelsource functions:

```
this.inv_levelsource.of_SetPictureColumn(1, "1")
this.inv_levelsource.of_SetSelectedPictureColumn  &
    (1, "2")
this.inv_levelsource.of_SetPictureColumn(2, "4")
this.inv_levelsource.of_SetSelectedPictureColumn  &
    (2, "5")
```

v **To use DataWindow columns to specify TreeView pictures:**

1 Establish database or DataWindow columns and populate them with bitmap names (String or Character data type) or picture index specifications (Integer data type).

2 Enable the TreeView level source service:

```
this.of_SetLevelSource(TRUE)
```

3 Specify the appropriate pictures by calling n_cst_tvsrv_levelsource functions, passing the DataWindow column names:

```
this.inv_levelsource.of_SetPictureColumn  &
    (1, 'picture_name')
```

```
this.inv_levelsource.of_SetSelectedPictureColumn  &
   (1, 'picture_overlay')
```

Bitmaps must exist
The files named in the retrieved rows must exist in a directory accessible to the application.

For more information on TreeView pictures, see the *PowerBuilder User's Guide*.

Coordinating a TreeView and other controls

One of the most popular uses for TreeViews is to perform coordinated processing with a ListView. The Microsoft Explorer is an example of this type of usage.

Another powerful possibility is to coordinate processing between a TreeView and a DataWindow.

v **To coordinate a TreeView and a ListView:**

1 Create DataWindow objects to display information for all levels of the TreeView (this example uses region, state, customer, and employee information from the PFC example database).

2 Create a window that has a TreeView based on u_tvs and a ListView based on u_lvs.

3 Define pictures for the TreeView and ListView.

4 Enable the TreeView level source service:

```
this.of_SetLevelSource(TRUE)
```

5 Register level source information for all TreeView levels (this example is from the TreeView Constructor event):

```
this.inv_levelsource.of_Register(1,  &
    "sales_regions_region", "", "d_region",  &
          SQLCA, "")
this.inv_levelsource.of_SetPictureColumn(1, "1")
this.inv_levelsource.of_SetSelectedPictureColumn  &
       (1, "7")

this.inv_levelsource.of_Register(2,  &
   "states_state_name",  &
      ":parent.1.sales_regions_region", &
         "d_regionstate", SQLCA, "")
this.inv_levelsource.of_SetPictureColumn(2, "2")
this.inv_levelsource.of_SetSelectedPictureColumn  &
       (2, "7")
```

```
this.inv_levelsource.of_Register(3,  &
   "customer_company_name",  &
       ":parent.2.sales_regions_region,  &
          :parent.1.states_state_id",  &
             "d_regionstatecust", SQLCA, "")
this.inv_levelsource.of_SetPictureColumn(3, "3")
this.inv_levelsource.of_SetSelectedPictureColumn  &
       (3, "7")

this.inv_levelsource.of_Register(4,  &
   "employee_emp_lname", ":parent.1.customer_id", &
      "d_regionstatecustrep", SQLCA, "")
this.inv_levelsource.of_SetPictureColumn(4, "4")
this.inv_levelsource.of_SetSelectedPictureColumn  &
   (4, "7")

this.inv_levelsource.of_Register(5,  &
   "order_id_string", ":parent.2.customer_id,  &
       :parent.1.employee_emp_id", &
         "d_regionstatecustrepord", SQLCA, "")
this.inv_levelsource.of_SetPictureColumn(5, "5")
this.inv_levelsource.of_SetSelectedPictureColumn  &
   (5, "7")
```

6 Call the pfc_Populate event:

```
this.Event pfc_Populate(0)
```

7 Extend the pfc_Retrieve event:

```
Any la_args[20]
Integer li_level

IF IsValid(inv_levelsource) THEN
      li_level = of_GetNextLevel(al_parent)
      inv_levelsource.of_GetArgs(al_parent,  &
         li_level, la_args)
END IF
Return of_Retrieve(al_parent, la_args, ads_data)
```

8 Extend the TreeView's SelectionChanged event to call the ListView's pfc_Populate event:

```
lv_1.Event pfc_Populate()
```

9 Enable ListView services and specify processing options (this example is from the ListView Constructor event):

```
this.of_SetDataSource(TRUE)
this.of_SetSort(TRUE)
this.inv_sort.of_SetColumnHeader(TRUE)
this.of_SetRMBMenu(TRUE)
```

10 Override the ListView’s pfc_Populate event:

```
Integer   li_RC, li_level
Long  ll_handle
String  ls_dataobject
String  ls_labelcolumn, ls_picturecolumn
TreeViewItem   ltvi_selecteditem
n_tr  ltr_obj
n_cst_tvsrvattrib   lnv_tvattrib

// Display current tree item children in the LV
ll_handle = tv_1.FindItem(CurrentTreeItem!, 0)
tv_1.GetItem(ll_handle, ltvi_selecteditem)

li_level = ltvi_selecteditem.Level + 1

// Normal registration
ls_dataobject =  &
   tv_1.inv_levelsource.of_GetDataObject(li_level)
ls_labelcolumn =  &
   tv_1.inv_levelsource.of_GetLabelColumn(li_level)
ls_picturecolumn = &
   tv_1.inv_levelsource.of_GetPictureColumn  &
     (li_level)
tv_1.inv_levelsource.of_GetTransObject(li_level,  &
    ltr_obj)

// Level 3 registration.
tv_1.inv_levelsource.of_GetLevelAttributes  &
   (li_level, lnv_tvattrib)

// Set the ListView items
Choose Case ltvi_selecteditem.Level
Case 1
  li_RC =  lv_1.inv_datasource.of_Register  &
    (ls_labelcolumn, ls_dataobject, ltr_obj)
  li_RC =  &
    lv_1.inv_datasource.of_SetPictureColumn  &
      (ls_picturecolumn)
```

```
Case 2
  li_RC = lv_1.inv_datasource.of_Register  &
    (ls_labelcolumn, ls_dataobject, ltr_obj)
  li_RC =  &
   lv_1.inv_datasource.of_SetPictureColumn  &
     (ls_picturecolumn)
Case 3
  li_RC =  lv_1.inv_datasource.of_Register  &
    (lnv_tvattrib.is_labelcolumn, &
      lnv_tvattrib.is_dataobject, &
        lnv_tvattrib.itr_obj)
  li_RC =  &
    lv_1.inv_datasource.of_SetPictureColumn  &
     (lnv_tvattrib.is_picturecolumn)
Case 4
  li_RC = lv_1.inv_datasource.of_Register  &
   (ls_labelcolumn, ls_dataobject, ltr_obj)
  li_RC =  &
    lv_1.inv_datasource.of_SetPictureColumn  &
     (ls_picturecolumn)
Case 5  // Not in the tree so register normally
  li_RC = lv_1.inv_datasource.of_Register  &
    ("product_description",  &
       "d_regionstatecustrepordítm", SQLCA)
  li_RC =  &
    lv_1.inv_datasource.of_SetPictureColumn  &
      ("product_picture_name")
End Choose
// Add all the visible columns of the datasource
// to the report view.
lv_1.inv_datasource.of_RegisterReportColumn()

Return Super::Event pfc_Populate()
```

11 Extend the ListView's pfc_Retrieve event:

```
Long   ll_handle
Any   la_args[20]
TreeViewItem  tvi_item

ll_handle = tv_1.FindItem(CurrentTreeItem!, 0)
tv_1.GetItem(ll_handle, ltvi_item)

If ltvi_item.Level < 5 Then
      tv_1.inv_levelsource.of_GetArgs  &
      (ll_handle, (ltvi_item.Level + 1), la_Args)
```

```
Else
       la_Args[1] = Integer(ltvi_item.Label)
End If

Return of_Retrieve(la_args, ads_data)
```

12 Add code to the ListView's DoubleClicked event to coordinate display with the TreeView:

```
Integer  li_level
Long  ll_currenttvitem, ll_selectedtreehandle
String  ls_lvlabel
ListViewItem  llvi_selectedlvitem
TreeViewItem  ltvi_newtreeitem
TreeViewItem ltvi_startingtreeitem

// Get the ListView item that was doubleclicked.
this.GetItem(index, llvi_selectedlvitem)
ls_lvlabel = llvi_selectedlvitem.label

// Determine which TreeView item is currently
// selected and get it.
ll_currenttvitem = tv_1.FindItem  &
  (CurrentTreeItem!, 0)
tv_1.GetItem(ll_currenttvitem,  &
  ltvi_startingtreeitem)

// Set a local variable to the level of the
// currently selected TreeView item.
li_level = ltvi_startingtreeitem.level

// Determine if the currently selected TreeView
// item has been expanded. If it hasn't, expand it.
// (expanding also populates). This loads the
//  TreeView with the ListView information.
IF ltvi_startingtreeitem.expanded = FALSE THEN
       tv_1.ExpandItem(ll_currenttvitem)
END IF

// Get the handle of the TreeView item that
// corresponds to the ListView item that was
// doubleclicked.
ll_selectedtreehandle =  &
  tv_1.inv_base.of_FindItem ("label", ls_lvlabel, &
    ll_currenttvitem, (li_level + 1), TRUE, TRUE)

// Get the state information of the TreeView item
// that corresponds to the ListView item that was
// doubleclicked.
tv_1.GetItem(ll_selectedtreehandle,  &
  ltvi_newtreeitem)

// Select and Expand the selected TreeView item.
IF ltvi_newtreeitem.expanded = FALSE THEN
       tv_1.SelectItem(ll_selectedtreehandle)
       tv_1.ExpandItem(ll_selectedtreehandle)
END IF
```

v **To coordinate a TreeView and a DataWindow control:**

1 Create DataWindow objects to display information for all levels of the TreeView (this example uses region, state, customer, and employee information from the PFC example database).

2 Create a window that has a TreeView based on u_tvs and a DataWindow based on u_dw.

3 Define pictures for the TreeView.

4 Enable the TreeView level source service:

```
this.of_SetLevelSource(TRUE)
```

5 Register level source information for all TreeView levels (this example is from the TreeView Constructor event):

```
this.inv_levelsource.of_Register(1,  &
  "sales_regions_region", "", "d_region",  &
     SQLCA, "")
this.inv_levelsource.of_SetPictureColumn(1, "1")
this.inv_levelsource.of_SetSelectedPictureColumn  &
  (1, "7")

this.inv_levelsource.of_Register(2, &
  "states_state_name",  &
    ":parent.1.sales_regions_region",  &
      "d_regionstate", SQLCA, "")
this.inv_levelsource.of_SetPictureColumn(2, "2")
this.inv_levelsource.of_SetSelectedPictureColumn  &
  (2, "7")

this.inv_levelsource.of_Register(3,  &
   "customer_company_name",  &
     ":parent.2.sales_regions_region,  &
       :parent.1.states_state_id",  &
         "d_regionstatecust", SQLCA, "")
```

```
this.inv_levelsource.of_SetPictureColumn(3, "3")
this.inv_levelsource.of_SetSelectedPictureColumn  &
  (3, "7")

this.inv_levelsource.of_Register(4,  &
  "employee_emp_lname", ":parent.1.customer_id", &
    "d_regionstatecustrep", SQLCA, "")
this.inv_levelsource.of_SetPictureColumn(4, "4")
this.inv_levelsource.of_SetSelectedPictureColumn  &
  (4, "7")

this.inv_levelsource.of_Register(5,  &
  "order_id_string", ":parent.2.customer_id,  &
    :parent.1.employee_emp_id", &
      "d_regionstatecustrepord", SQLCA, "")
this.inv_levelsource.of_SetPictureColumn(5, "5")
this.inv_levelsource.of_SetSelectedPictureColumn  &
  (5, "7")
```

6 Call the pfc_Populate event:

```
this.Event pfc_Populate(0)
```

7 Extend the pfc_Retrieve event:

```
Any la_args[20]
Integer li_level

IF IsValid(inv_levelsource) THEN
       li_level = of_GetNextLevel(al_parent)
       inv_levelsource.of_GetArgs(al_parent,  &
          li_level, la_args)
END IF
Return of_Retrieve(al_parent, la_args, ads_data)
```

8 Extend the TreeView's SelectionChanging event to reset the DataWindow control's DataObject property and populate it with the selected TreeView item:

```
n_ds  lds_datastore
TreeViewItemltvi_new
Long  ll_dsrow

// Get the DataStore and row for the new item
IF inv_levelsource.of_GetDataRow(newhandle,  &
    lds_datastore, ll_dsrow) = -1 THEN
       MessageBox("Error", &
       "Error in of_GetDataRow", Exclamation!)
END IF
```

```
// Set dw_1 to use the new DataStore
dw_1.Reset()
dw_1.DataObject = lds_datastore.DataObject

// Copy the row for the selected item
// in the DataStore.
lds_datastore.RowsCopy(ll_dsrow, ll_dsrow,  &
   Primary!, dw_1, 1, Primary!)

// Set status flag of new row to what
// it was in the TreeView level datasource.
// The new row is copied as NewModified!
CHOOSE CASE lds_datastore.GetItemStatus  &
  (ll_dsrow, 0, Primary!)
CASE New!
  dw_1.SetItemStatus(1, 0, Primary!, NotModified!)
CASE DataModified!
  dw_1.SetItemStatus(1, 0, Primary!, DataModified!)
CASE NotModified!
  dw_1.SetItemStatus(1, 0, Primary!, DataModified!)
  dw_1.SetItemStatus(1, 0, Primary!, NotModified!)
END CHOOSE
```

Printing a TreeView

U_tvs allows you to print TreeViews, optionally displaying a cancel dialog box and customized level pictures.

v **To print a TreeView:**

1 Enable the TreeView print service:

```
tv_deptemp.of_SetPrint(TRUE)
```

2 Call the of_PrintTree function:

```
tv_deptemp.inv_print.of_PrintTree()
```

Using the u_rte RichTextEdit control

You use the PowerBuilder RichTextEdit control to enhance an application with word processing capabilities. The PFC u_rte control makes it easier for you to work with a RichTextEdit control. U_rte allows you to:

- Display new documents, optionally inserting them into the current document
- Insert pictures into documents, optionally displaying a dialog box prompting the user for the filename

- Print documents
- Add Find and Replace capabilities to a RichTextEdit control
- Control text properties

For complete information on the PowerBuilder RichTextEdit control, see the *PowerBuilder User's Guide* and *Application Techniques*.

Displaying rich text documents

U_rte provides events that allow the user to specify the rich text document to open:

- **pfc_Open** Replaces the current document with the selected document, prompting the user before discarding the current document
- **pfc_InsertFile** Inserts the selected document into the current document. The RichTextEdit control replaces the current selection when the file is inserted

You can also populate the RichTextEdit control by calling the PowerScript InsertDocument function.

v **To display a document (replacing the current document):**

- Call the pfc_Open event:

```
rte_doc.Event pfc_Open()
```

v **To insert a document into the current document:**

- Call the pfc_InsertFile function:

```
rte_doc.Event pfc_InsertFile()
```

The InsertDocument PowerScript function
You can also call the PowerScript InsertDocument function to display a document. If you use this function to display a file in an empty control, you must also call the of_SetFileName function to specify the name of the file associated with the RichTextEdit control.

Inserting pictures

U_rte provides an event that displays a dialog box for the user to choose a bitmap to insert at the current cursor position. The RichTextEdit control replaces the current selection when the bitmap is inserted.

v **To display the Insert Picture dialog box:**

- Call the pfc_InsertPicture event:

```
rte_doc.Event pfc_InsertPicture()
```

Tracking the picture's filename
If you want to track the filename of the inserted picture, call the of_InsertPicture function instead of the pfc_InsertPicture event.

Printing rich text documents

U_rte provides events that allow you to print the data in RichTextEdit controls. You can:

- Display a Print dialog box, allowing you to choose options before printing

  PFC uses the s_printdlgattrib structure to pass properties to the n_cst_platform of_PrintDlg function. You can use the pfc_PrePrintDlg event to further customize the contents of the Print dialog box by modifying elements in the s_printdlgattrib structure.

  The values in the s_printdlgattrib structure reflect selected RichTextEdit Print properties (such as collate, page numbers, and number of copies).

- Print a RichTextEdit control without displaying the Print dialog box

U_rte also provides functions that allow you to control the printing of page numbers.

v **To display the Print dialog box:**

1. (Optional) Add code to the pfc_PrePrintDlg event to modify the information used by the pfc_PrintDlg function (this example provides a default for the number of copies specification):

   ```
   astr_printdlg.l_copies = 1
   ```

2. Call the pfc_Print event:

   ```
   rte_doc.Event pfc_Print()
   ```

v **To print a RichTextEdit control without displaying the Print dialog box:**

- Call the pfc_PrintImmediate event:

  ```
  rte_doc.Event pfc_PrintImmediate()
  ```

v **To control the printing of page numbers:**

1. (Optional) Specify the page number upon which page numbers should first appear. For example, many styles suppress the page number on the first page of a document:

   ```
   rte_doc.of_SetStartPageNumber(2)
   ```

2 Identify the name of the field into which PFC places the page number (this example assumes an input field named PAGENUM):

```
rte_doc.of_SetPageInputField("PAGENUM")
```

v **To print continuous pages when sharing data with a DataWindow or DataStore:**

1 Perform all the steps necessary for the RichText file to share data with the DataWindow or DataStore:

```
ids_empdata = CREATE n_ds

ids_empdata.DataObject = "d_sharerte"
ids_empdata.of_SetTransObject(SQLCA)
IF ids_empdata.Retrieve() = -1 THEN
  MessageBox("Retrieve", "Retrieve error")
END IF

rte_doc.DataSource(ids_empdata)
```

2 Call the of_SetContinuousPages function:

```
rte_doc.of_SetContinuousPages(TRUE)
```

3 Print the document:

```
rte_doc.Event pfc_Print( )
```

For more information on sharing data between a RichTextEdit control and a DataWindow or DataStore, see *Application Techniques*.

Using the RTE find service

U_rte features a find and replace service that you can use to enhance a RichTextEdit control. Once the service is enabled, PFC displays Find and Replace dialog boxes when the user selects Edit>Find or Edit>Replace from the menu bar of a menu that descends from the PFC m_master menu and the RichTextEdit control has focus.

You can also display Find and Replace dialog boxes programmatically.

v **To display the Find dialog box:**

1 Enable the Find service:

```
rte_doc.of_SetFind(TRUE)
```

2 Call the pfc_FindDlg event:

```
rte_doc.Event pfc_FindDlg()
```

v **To display the Replace dialog box:**

1 Enable the Find service:

```
rte_doc.of_SetFind(TRUE)
```

2 Call the pfc_ReplaceDlg event:

```
rte_doc.Event pfc_ReplaceDlg()
```

Controlling text properties

U_rte allows you to access properties for selected text in a RichTextEdit control. This feature allows you to change a single property at a time, leaving the other properties as is. This differs from the PowerScript SetTextStyle function, which requires that you specify all possible text properties.

v **To set text properties:**

1 (Optional) Establish a mechanism that allows the user to specify text properties (the example ahead uses checkboxes).

2 Call the of_SetTextStyle*xxx* functions to set text properties:

```
SetRedraw(FALSE)

rte_doc.of_SetTextStyleBold(cbx_bold.Checked)
rte_doc.of_SetTextStyleItalic(cbx_italic.Checked)
rte_doc.of_SetTextStyleUnderline &
  (cbx_underline.Checked)
rte_doc.of_SetTextStyleStrikeout &
  (cbx_strikeout.Checked)
rte_doc.of_SetTextStyleSubscript &
  (cbx_subscript.Checked)
rte_doc.of_SetTextStyleSuperscript &
  (cbx_superscript.Checked)

SetRedraw(TRUE)
```

v **To access text properties:**

1 (Optional) Establish a mechanism that displays text properties (the example ahead uses checkboxes).

2 Call the of_GetTextStyle function:

```
n_cst_textstyleattrib lnv_style

rte_doc.of_GetTextStyle &
  (lnv_style)
cbx_bold.Checked = lnv_style.ib_bold
cbx_italic.Checked = lnv_style.ib_italic
```

```
cbx_underline.Checked = &
  lnv_style.ib_underlined
cbx_strikeout.Checked = lnv_style.ib_strikeout
cbx_subscript.Checked = lnv_style.ib_subscript
cbx_superscript.Checked = &
  lnv_style.ib_superscript
```

Using the u_oc OLE control

PFC provides functions and events that offer basic control over an OLE control based on u_oc. In addition to standard support for editable controls (such as cut, copy, paste, and right-mouse button support), u_oc includes Paste Special functionality, support for in-place and off-site activation, and update links functionality.

For complete information on programming with the OLE control, see *Application Techniques*.

Displaying the Insert Object dialog box

You display the Insert Object dialog box to change the object or the server application.

v **To display the Insert Object dialog box:**

1 Create a window that has an OLE control based on u_oc.

2 Call the pfc_InsertObject event:

```
ole_1.Event pfc_InsertObject()
```

Activating an object in place

When you activate an object in place, the user interacts with the object inside the PowerBuilder application's window.

1 Create a window that has an OLE control based on u_oc.

2 Call the pfc_EditObject event:

```
ole_1.Event pfc_EditObject()
```

Activating the object offsite

When you activate an object offsite, the server application opens and the object becomes an open document in the server's window.

1 Create a window that has an OLE control based on u_oc.

2 Call the pfc_OpenObject event:

```
ole_1.Event pfc_OpenObject()
```

Updating the linked object

When you update a linked object, PowerBuilder attempts to find a file linked to an OLE container. If the linked file is not found, a dialog displays and lets the user bring up a second dialog for finding the file or changing the link.

1 Create a window that has an OLE control based on u_oc.

2 Call the pfc_UpdateLinks event:

```
ole_1.Event pfc_UpdateLinks()
```

Using the u_tab Tab control and the u_tabpg user object

Many current applications use Tab controls and tab pages to enhance their user interface. The u_tab Tab control and the u_tabpg user object provide basic PFC functionality. And they both implement the resize service, which you enable differently depending on how the tab is defined.

About Tab controls and tab pages

Tab control A Tab control is a control that you place in a window or user object that contains tab pages. Part of the area in the Tab control is for the tabs associated with the tab pages. Any space left is occupied by the tab pages themselves.

PFC provides the u_tab Tab control, which you can use as an ancestor for Tab controls.

Tab page A tab page contains other controls and is one of several pages within a Tab control. All tab pages in a Tab control occupy the same area of the control, and only one is visible at a time. The active tab page covers the other tab pages. There are different ways to approach tab page definition. You can define:

- **An embedded tab page** In the Window or User Object painter, Select Insert>TabPage from the Tab control’s pop-up menu and add controls to those pages. An embedded tab page is of class UserObject but is not reusable.
- **A tab page user object** In the User Object painter, create a custom visual user object and add the controls that will display on the tab page. To add a tab page user object to a Tab control, Select Insert>User Object from the Tab control’s pop-up menu. A tab page defined as an independent user object is reusable.

  PFC provides the u_tabpg custom visual user object, which you can use as the ancestor for tab pages.

You can mix and match the two methods—one Tab control can contain both embedded tab pages and tab page user objects. But non-PFC tab pages do not have support for PFC features, such as resizing and the message router.

For more information on programming with Tab controls and tab pages, see *Application Techniques*.

Using u_tab

When using u_tab, you always:

- Work with a descendant of u_tab
- Create the complete Tab control (including tab pages) in the User Object painter

v **To create a Tab control:**

1 Create a user object based on u_tab.

2 Add embedded tab pages and tab page user objects. For embedded tab pages, you add controls as necessary. Tab page user objects must be completely defined; you cannot add or modify controls from within the Tab control.

3 Create, override, and extend tab-page events and functions as necessary.

4 Create, override, and extend tab-level events and functions as necessary. Then call events in the tab pages, optionally defining user events on the Tab control that call the tab page events.

5 Add the user object to a window.

Using u_tabpg

When using u_tabpg, you always work with a descendant of u_tabpg within the User Object painter.

v **To create a tab page:**

1 Create a user object based on u_tabpg.

2 Add controls as necessary.

3 Add PowerScript code for the controls as necessary.

4 Create, override, and extend tab page events and functions as necessary.

5 Add the tab page user object to one of the following:

  - U_tab-based user object
  - Standard visual user object of type Tab
  - Tab control within the Window painter

Using the resize service with u_tab descendants

You can use the resize service to add dynamic resize capabilities to tab pages.

How you enable the resize service differs depending on whether the Tab control contains embedded tab pages or tab page user objects based on u_tabpg. If a Tab control contains both embedded tab pages and tab page user objects, you can combine the first two procedures that follow.

When using the Create on Demand option for tab page user objects, you must also perform the third procedure to ensure that controls display properly.

v **To enable the resize service for a Tab control when using embedded tab pages:**

1. Use the User Object painter to create a Tab control based on u_tab.
2. Define embedded tab pages.
3. Enable the resize service for the Tab control (this example is from the Constructor event):

   ```
   this.of_SetResize(TRUE)
   ```

4. Register controls within the tab pages:

   ```
   this.inv_resize.of_Register &
     (this.tabpage_1.mle_1, &
        this.inv_resize.SCALETOBOTTOM)
   this.inv_resize.of_Register &
     (this.tabpage_2.dw_1, &
        this.inv_resize.SCALETOBOTTOM)
   ```

5. Add the user object to a window.
6. Enable the resize service for the window and register affected controls (this example is from the window’s Constructor event):

   ```
   this.of_SetResize(TRUE)
   this.inv_resize.of_Register(tab_1, "Scale")
   this.inv_resize.of_Register &
     (cb_cancel, &
        this.inv_resize.FIXEDTOBOTTOM)
   this.inv_resize.of_Register &
     (cb_ok, this.inv_resize.FIXEDTOBOTTOM)
   this.inv_resize.of_SetMinSize &
     (this.width - 100, this.height - 100)
   ```

v **To enable the resize service for a Tab control when using tab page user objects based on u_tabpg:**

1 Use the User Object painter to create a tab page based on u_tabpg.

2 Enable the resize service for the tab page (this example is from the Constructor event):

```
this.of_SetResize(TRUE)
```

3 Register controls within the tab page:

```
this.inv_resize.of_Register &
  (this.dw_1, 0, 0, 100, 100))
```

4 Add the tab page user object to a Tab control.

If there are no embedded tab pages
If a Tab control contains only tab page user objects based on u_tabpg (but not embedded tab pages), you need not enable the resize service for the Tab control.

5 If the Tab control is a user object, add it to a window.

6 Enable the resize service for the window and register affected controls (this example is from the Constructor event):

```
this.of_SetResize(TRUE)
this.inv_resize.of_Register &
  (tab_1, 0, 0, 100, 100)
this.inv_resize.of_Register &
  (cb_cancel, &
this.inv_resize.FIXEDTOBOTTOM)
this.inv_resize.of_Register &
  (cb_ok, &
this.inv_resize.FIXEDTOBOTTOM)
this.inv_resize.of_SetMinSize &
  (this.width - 100, this.height - 100)
```

v **To use the resize service with tab page user objects that have the Create on Demand property:**

1 In the User Object painter, display the Position tab of the object’s property sheet to determine what the size will be at creation time.

2 Specify the original size by calling the of_SetOrigSize function (this example is from the tab page’s Constructor event):

```
this.inv_resize.of_SetOrigSize(1637, 457)
```

3 Register tab page controls:

```
this.inv_resize.of_Register &
  (this.dw_1, 0, 0, 100, 100))
```

4 Trigger the Resize event on the UserObject (this example is from the Tab control's Constructor event):

```
// The Resize event moves registered
// objects, as appropriate.
this.tabpage_1.TriggerEvent(Resize!)
```

Using custom visual user objects

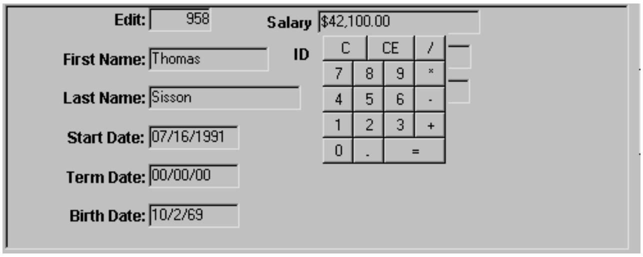
PFC includes custom visual user objects that you can use to enhance applications:

- **A calculator:** u_calculator
- **A calendar:** u_calendar
- **A splitbar:** u_st_splitbar
- **A progress bar:** u_progressbar

Using the calculator control

You use u_calculator (the PFC calculator control) to provide any of the following:

- Drop-down calculator for numeric columns in a DataWindow:



Edit: 958

Salary $42,100.00

First Name: Thomas

ID

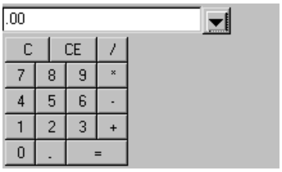
Last Name: Sisson

Start Date: 07/16/1991

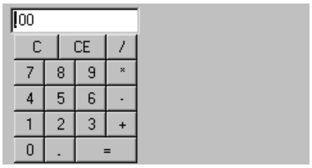
Term Date: 00/00/00

Birth Date: 10/2/69

- Drop-down calculator for numeric or decimal values in an EditMask:



- Standalone calculator for use with an EditMask:



Users make calculations using the drop-down calculator. To enter numbers they either click the calculator's buttons or use the numeric keypad with NumLock turned on. The calculator automatically enters calculation results into the associated field.

The PFC calculator control includes functions that allow you to control certain aspects of calculator behavior. For example, you call the of_SetCloseOnClick function to control whether the drop-down calculator closes when the user clicks the equal sign.

Using a drop-down calculator with a DataWindow control

The PFC calculator control works with DataWindow columns that have a numeric or decimal data type and are registered with u_calculator.

Depending on the registration option, the calculator displays when a registered column gets focus, when the user clicks the drop-down arrow, or when your code calls the pfc_DDCalculator event.

Controlling the visual cue To control the visual cue that displays in a DataWindow column for which the drop-down calculator is enabled, you supply an argument to the of_Register function:

Argument	Result
NONE	If the column uses the: • DropDownListBox edit style, the calculator displays automatically when the column gets the focus • Edit or EditMask edit style, the calculator displays when you call the pf_DDCalculator event

Argument	Result
DDLB	Of_Register converts all registered columns to the DropDownListBox edit style. The calculator displays when the user clicks the down arrow, which disappears when the calculator displays
DDLB_WITHARROW	Of_Register converts all registered columns to the DropDownListBox edit style. The calculator displays when the user clicks the down arrow, which remains when the calculator displays

Column edit styles When the service converts columns to the DropDownListBox edit style, properties that do not apply to DropDownListBoxes are lost. Because of this, you typically use the drop-down calculator with columns that already use the DropDownListBox edit style, calling of_Register passing NONE.

v **To use a drop-down calculator with DataWindow columns:**

1 Find out which numeric or decimal DataWindow columns are appropriate for use with a drop-down calculator. For example, a salary column might use a calculator for use when determining raises. These columns must use the DropDownListBox, Edit, or EditMask edit style.

2 Place a u_dw-based DataWindow control on the window or user object.

3 Enable the drop-down calculator by calling the u_dw of_SetDropdownCalculator function (this example is from a DataWindow Constructor event):

```
this.of_SetDropDownCalculator(TRUE)
```

4 Register columns one by one or all at once by calling the of_Register function. Of_Register includes an optional argument specifying the drop-down style:

```
this.iuo_calculator.of_Register("salary", &
      this.iuo_calculator.NONE)
```

5 Call additional functions as necessary to customize calculator behavior:

```
this.iuo_calculator.of_SetCloseOnClick(FALSE)
this.iuo_calculator.of_SetInitialValue(TRUE)
```

Displaying the calculator programmatically

You can also display the drop-down calculator programmatically. This works with all of_Register options and is required for Edit and EditMask columns when using the NONE option.

v **To display the drop-down calculator programmatically:**

1 Place a u_dw-based DataWindow control on the window or user object.

2 Enable the drop-down calculator by calling the u_dw of_SetDropdownCalculator function (this example is from a DataWindow Constructor event):

```
this.of_SetDropDownCalculator(TRUE)
```

3 Register columns to be displayed programmatically by calling the of_Register function. Of_Register includes an argument specifying the drop-down style. Programmatic display works best with the NONE style but can be used with any drop-down style:

```
this.iuo_calculator.of_Register("salary", &
   this.iuo_calculator.NONE)
```

4 Define a user event or visual control (such as a command button) that sets focus in the DataWindow control and calls the u_dw pfc_DDCalculator event:

```
IF dw_1.SetColumn("salary") = 1 THEN
  dw_1.Event pfc_DDCalculator( )
END IF
```

Using a drop-down calculator with an EditMask control

You can use a drop-down calculator with EditMask controls that use the numeric or decimal option type.

v **To use a drop-down calculator with an EditMask control:**

1 Place a u_em-based EditMask control on the window or user object.

2 Enable the drop-down calculator by calling the u_em of_SetDropdownCalculator function (this example is from an EditMask Constructor event):

```
this.of_SetDropDownCalculator(TRUE)
```

3 Call additional functions as necessary to customize calculator behavior:

```
this.iuo_calculator.of_SetCloseOnClick(FALSE)
this.iuo_calculator.of_SetInitialValue(TRUE)
```

4 Define a user event or visual control (such as a picture button) that displays the drop-down calculator by calling the u_em pfc_DDCalculator event:

```
em_1.Event pfc_DDCalculator( )
```

Using a standalone calculator

You can create a standalone calculator by placing u_calculator directly on a window or user object.

v **To create a standalone calculator:**

1 Place a u_em-based EditMask control on the window or user object.

2 Place an instance of u_calculator on the window or user object.

3 Associate the drop-down calculator with the EditMask by calling the u_calculator of_SetRequestor function (this example is from a u_calculator instance's Constructor event):

```
this.of_SetRequestor(parent.em_1)
```

Setting calculator options

The PFC calculator control provides options that you can set to control calculator behavior:

- **Close on click** Controls whether the drop-down calculator closes when the user clicks the equal sign:

```
this.of_SetDropDownCalculator(TRUE)
...
this.iuo_calculator.of_SetCloseOnClick(TRUE)
```

- **Initial value** Controls whether the calculator initializes blank fields with a zero when it first displays:

```
this.of_SetDropDownCalculator(TRUE)
...
this.iuo_calculator.of_SetInitialValue(TRUE)
```

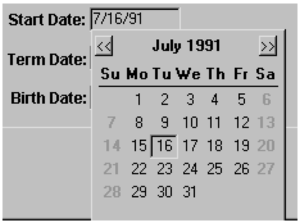
Use the Constructor event
You typically call the functions that control these behaviors in the u_dw or u_em Constructor event.

The examples in this discussion are from the Constructor event of a u_dw-based DataWindow control.

Using the calendar control

You use u_calendar (the PFC calendar control) to provide a drop-down calendar for date values in any of the following:

- U_dw-based DataWindow control:

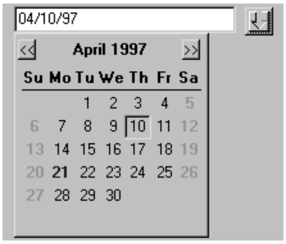


Start Date: 7/16/91
Term Date:
Birth Date:

July 1991

Su	Mo	Tu	We	Th	Fr	Sa
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

- U_em-based EditMask control:



04/10/97

April 1997

Su	Mo	Tu	We	Th	Fr	Sa
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30			

Users enter dates by clicking on the drop-down calendar, automatically entering the selected date in the associated field. They change months by clicking the >> and << buttons and can also navigate the calendar with keyboard arrow keys.

By default, all days appear with the same characteristics. You can specify a different color for Saturdays and Sundays as well as whether Saturdays and Sundays should appear bold. The calendar also allows you to highlight holidays and other marked days.

The PFC calendar control includes functions that allow you to control certain aspects of calendar behavior. For example, you call the of_SetInitialValue function to control whether the drop-down calendar initializes blank fields with the current date when it first displays.

Using a drop-down calendar with a DataWindow control

The drop-down calendar works with DataWindow columns that have a date data type and are registered with u_calendar.

Depending on the registration option, the calendar displays when a registered column gets focus, when the user clicks the drop-down arrow, or when your code calls the pfc_DDCalendar event.

Controlling the visual cue To control the visual cue that displays in a DataWindow column for which the drop-down calendar is enabled, you supply an argument to the of_Register function:

Argument	Result
NONE	If the column uses the: • DropDownListBox edit style, the calendar displays automatically when the column gets the focus • Edit or EditMask edit style, the calendar displays when you call the pf_DDCalendar event
DDLB	Of_Register converts all registered columns to the DropDownListBox edit style. The calendar displays when the user clicks the down arrow, which disappears when the calendar displays
DDLB_WITHARROW	Of_Register converts all registered columns to the DropDownListBox edit style. The calendar displays when the user clicks the down arrow, which remains when the calendar displays

Column edit styles When the service converts columns to the DropDownListBox edit style, properties that do not apply to DropDownListBoxes are lost. So you typically use the drop-down calendar with columns that already use the DropDownListBox edit style, calling of_Register passing NONE.

v **To use a drop-down calendar with DataWindow columns:**

1 Find which date DataWindow columns are appropriate for use with a drop-down calendar. For example, a requested delivery date column might use a drop-down calendar. These columns must use the DropDownListBox, Edit, or EditMask edit style.

2 Place a u_dw-based DataWindow control on the window or user object.

3 Enable the drop-down calendar by calling the u_dw of_SetDropdownCalendar function (this example is from a DataWindow Constructor event):

```
this.of_SetDropDownCalendar(TRUE)
```

4 Register columns one by one or all at once by calling the of_Register function. Of_Register includes an optional argument specifying the drop-down style:

```
this.iuo_calendar.of_Register("salary" &
   this.iuo_calendar.NONE)
```

5 (Optional) Establish the font style and color for weekend days:

```
this.iuo_calendar.of_SetSaturdayBold(TRUE)
this.iuo_calendar.of_SetSaturdayColor &
  (RGB(0, 255, 0))
this.iuo_calendar.of_SetSundayBold(TRUE)
this.iuo_calendar.of_SetSundayColor &
  (RGB(0, 255, 0))
```

6 (Optional) Establish a list of holidays with their font style and color (this example shows holidays for one year only):

```
Date  ld_holidays[11]

ld_holidays[1] = 1997-01-01
ld_holidays[2] = 1997-02-17
ld_holidays[3] = 1997-04-21
ld_holidays[4] = 1997-05-26
ld_holidays[5] = 1997-07-04
ld_holidays[6] = 1997-09-01
ld_holidays[7] = 1997-10-13
ld_holidays[8] = 1997-11-27
ld_holidays[9] = 1997-11-28
ld_holidays[10] = 1997-12-25
ld_holidays[11] = 1997-12-26
...
this.iuo_calendar.of_SetHoliday(ld_holidays)
this.iuo_calendar.of_SetHolidayBold(TRUE)
this.iuo_calendar.of_SetHolidayColor &
  (RGB(0, 255, 0))
```

7 (Optional) Establish a list of marked days with their font style and color:

```
Date  ld_holidays[11], ld_marked_days[12]

ld_marked_days[1] = 1996-06-13
ld_marked_days[2] = 1996-03-16
ld_marked_days[3] = 1996-09-23
ld_marked_days[4] = 1996-09-14
ld_marked_days[5] = 1997-06-13
ld_marked_days[6] = 1997-03-16
ld_marked_days[7] = 1997-09-23
ld_marked_days[8] = 1997-09-14
ld_marked_days[9] = 1998-06-13
ld_marked_days[10] = 1998-03-16
ld_marked_days[11] = 1998-09-23
ld_marked_days[12] = 1998-09-14
...
```

```
this.iuo_calendar.of_SetMarkedDay(ld_marked_days)
this.iuo_calendar.of_SetMarkedDayBold(TRUE)
this.iuo_calendar.of_SetMarkedDayColor &
  (RGB(255, 0, 0))
```

Ensuring consistency
To ensure that all users see the same calendar display, define display characteristics, holidays, and marked days in the u_calendar (extension-level object) Constructor event.

8 (Optional) Call additional functions as necessary to customize calendar behavior:

```
this.iuo_calendar.of_SetAlwaysRedraw(TRUE)
this.iuo_calendar.of_SetInitialValue(TRUE)
```

Displaying the calendar programmatically

You can also display the drop-down calendar programmatically. This works with all of_SetRegister options and is required for Edit and EditMask columns when using the NONE option.

v **To display the drop-down calendar programmatically:**

1 Place a u_dw-based DataWindow control on the window or user object.

2 Enable the drop-down calendar by calling the u_dw of_SetDropdownCalendar function (this example is from a DataWindow Constructor event):

```
this.of_SetDropDownCalendar(TRUE)
```

3 Register columns to be displayed programmatically by calling the of_Register function. Of_Register includes an argument specifying the drop-down style. Programmatic display works best with the NONE style but can be used with any drop-down style:

```
this.iuo_calendar.of_Register("start_date" &
  this.iuo_calendar.NONE)
```

4 Define a user event or visual control (such as a command button) that sets focus in the DataWindow control and calls the u_dw pfc_DDCalendar event:

```
IF dw_1.SetColumn("start_date") = 1 THEN
  dw_1.Event pfc_DDCalendar( )
END IF
```

Using a drop-down calendar with an EditMask control

You can use a drop-down calculator with EditMask controls that use the date option type.

v **To use a drop-down calendar with an EditMask control:**

1 Place a u_em-based EditMask control on the window or user object.

2 Enable the drop-down calendar by calling the u_em of_SetDropdownCalendar function (this example is from an EditMask Constructor event):

```
this.of_SetDropDownCalendar(TRUE)
```

3 (Optional) Establish the font style and color for weekend days:

```
this.iuo_calendar.of_SetSaturdayBold(TRUE)
this.iuo_calendar.of_SetSaturdayColor &
  (RGB(0, 255, 0))
this.iuo_calendar.of_SetSundayBold(TRUE)
this.iuo_calendar.of_SetSundayColor &
  (RGB(0, 255, 0))
```

4 (Optional) Establish a list of holidays with their font style and color (this example shows holidays for one year only):

```
Date  ld_holidays[11]

ld_holidays[1] = 1997-01-01
ld_holidays[2] = 1997-02-17
ld_holidays[3] = 1997-04-21
ld_holidays[4] = 1997-05-26
ld_holidays[5] = 1997-07-04
ld_holidays[6] = 1997-09-01
ld_holidays[7] = 1997-10-13
ld_holidays[8] = 1997-11-27
ld_holidays[9] = 1997-11-28
ld_holidays[10] = 1997-12-25
ld_holidays[11] = 1997-12-26
...
this.iuo_calendar.of_SetHoliday(ld_holidays)
this.iuo_calendar.of_SetHolidayBold(TRUE)
this.iuo_calendar.of_SetHolidayColor &
  (RGB(0, 255, 0))
```

5 (Optional) Establish a list of marked days with their font style and color:

```
Date  ld_holidays[11], ld_marked_days[12]

ld_marked_days[1] = 1996-06-13
ld_marked_days[2] = 1996-03-16
ld_marked_days[3] = 1996-09-23
ld_marked_days[4] = 1996-09-14
ld_marked_days[5] = 1997-06-13
```

```
ld_marked_days[6] = 1997-03-16
ld_marked_days[7] = 1997-09-23
ld_marked_days[8] = 1997-09-14
ld_marked_days[9] = 1998-06-13
ld_marked_days[10] = 1998-03-16
ld_marked_days[11] = 1998-09-23
ld_marked_days[12] = 1998-09-14
...
this.iuo_calendar.of_SetMarkedDay(ld_marked_days)
this.iuo_calendar.of_SetMarkedDayBold(TRUE)
this.iuo_calendar.of_SetMarkedDayColor &
(RGB(255, 0, 0))
```

6 (Optional) Call additional functions as necessary to customize calendar behavior:

```
this.iuo_calendar.of_SetAlwaysRedraw(TRUE)
this.iuo_calendar.of_SetInitialValue(TRUE)
```

7 Define a user event or visual control (such as a picture button) that displays the drop-down calendar by calling the u_em pfc_DDCalendar event:

```
em_1.Event pfc_DDCalendar( )
```

Establishing weekend display options

The PFC calendar control allows you to specify distinct colors and/or bold for Saturdays and Sundays.

v **To establish weekend display options:**

1 Specify whether Saturdays appear in bold by calling the of_SetSaturdayBold function:

```
this.iuo_calendar.of_SetSaturdayBold(TRUE)
```

2 Specify a color for Saturdays by calling the of_SetSaturdayColor function:

```
this.iuo_calendar.of_SetSaturdayColor &
   (RGB(0, 255, 0))
```

3 Specify whether Sundays appear in bold by calling the of_SetSundayBold function:

```
this.iuo_calendar.of_SetSundayBold(TRUE)
```

4 Specify a color for Sundays by calling the of_SetSundayColor function:

```
this.iuo_calendar.of_SetSundayColor &
   (RGB(0, 255, 0))
```

Establishing holidays and marked days

The PFC calendar control allows you to establish a set of holidays and a set of marked days.

You can specify distinct colors and/or bold for holidays and marked days.

Specify dates for more than one year
Users can navigate through multiple years in the PFC calendar control. You should specify holidays and marked days to handle as many years as needed.

v **To establish holidays and marked days:**

1 Establish arrays containing the lists of holidays and marked days:

```
Date  ld_holidays[11], ld_marked_days[12]

ld_holidays[1] = 1997-01-01
ld_holidays[2] = 1997-02-17
ld_holidays[3] = 1997-04-21
ld_holidays[4] = 1997-05-26
ld_holidays[5] = 1997-07-04
ld_holidays[6] = 1997-09-01
ld_holidays[7] = 1997-10-13
ld_holidays[8] = 1997-11-27
ld_holidays[9] = 1997-11-28
ld_holidays[10] = 1997-12-25
ld_holidays[11] = 1997-12-26

ld_marked_days[1] = 1996-06-13
ld_marked_days[2] = 1996-03-16
ld_marked_days[3] = 1996-09-23
ld_marked_days[4] = 1996-09-14
ld_marked_days[5] = 1997-06-13
ld_marked_days[6] = 1997-03-16
ld_marked_days[7] = 1997-09-23
ld_marked_days[8] = 1997-09-14
ld_marked_days[9] = 1998-06-13
ld_marked_days[10] = 1998-03-16
ld_marked_days[11] = 1998-09-23
ld_marked_days[12] = 1998-09-14
```

2 After enabling the calendar for the DataWindow or EditMask control, establish the list of holidays by calling the of_SetHoliday function:

```
this.iuo_calendar.of_SetHoliday(ld_holidays)
```

3 Establish holiday display options as necessary:

```
this.iuo_calendar.of_SetHolidayBold(TRUE)
this.iuo_calendar.of_SetHolidayColor &
  (RGB(0, 255, 0))
```

4 Establish the list of marked days by calling the of_SetMarkedDay function:

```
this.iuo_calendar.of_SetMarkedDay(ld_marked_days)
```

5 Establish marked day options as necessary:

```
this.iuo_calendar.of_SetMarkedDayBold(TRUE)
this.iuo_calendar.of_SetMarkedDayColor &
  (RGB(255, 0, 0))
```

Setting calendar options

The PFC calendar control provides options that you can set to control calendar behavior:

- **Close on click** Controls whether the drop-down calendar closes when the user clicks a date:

```
this.of_SetDropDownCalendar(TRUE)
...
this.iuo_calendar.of_SetCloseOnDClick(TRUE)
```

- **Close on double-click** Controls whether the drop-down calendar closes when the user double-clicks a date:

```
this.of_SetDropDownCalendar(TRUE)
...
this.iuo_calendar.of_SetCloseOnDClick(TRUE)
```

- **Date format** Controls the format of the date returned by the calendar:

```
this.of_SetDropDownCalendar(TRUE)
...
this.iuo_calendar.of_SetDateFormat("mm/dd/yy")
```

This must match the control’s date format
The of_SetDateFormat specification must match the DataWindow column’s edit format or the EditMask’s date mask.

- **Initialize date** Controls whether the calendar initializes blank fields with the current date when the calendar displays:

```
this.of_SetDropDownCalendar(TRUE)
...
this.iuo_calendar.of_SetInitialValue(TRUE)
```

Use the Constructor event
You typically call the functions that control these behaviors in the u_dw or u_em Constructor event.

Using the splitbar control

You use u_st_splitbar (the PFC splitbar control) to display a splitbar in windows and visual user objects. The splitbar separates two or more visual controls. By dragging the splitbar users can resize the surrounding controls dynamically.

Using the splitbar to separate controls

The splitbar control allows you to give windows a customizable interface, one of the fundamentals of good interface design. By dynamically resizing visual controls, users can easily control the information displayed. Typical uses for splitbars include:

- Between a TreeView and a ListView
- Between a TreeView and a DataWindow
- Between master and detail DataWindows

v **To use the splitbar control:**

1 Place an instance of u_st_splitbar on a window between two or more controls.

2 Move and resize the splitbar object and the surrounding objects until they relate appropriately. For example, a vertical splitbar between two objects should have the same height as the surrounding objects.

Setting the exact dimensions
Control the exact size and placement with the splitbar object’s property sheet.

3 Add code to the u_st_splitbar instance’s Constructor event to establish the line style and register the controls that resize when the user moves the splitbar:

```
this.of_Register(tv_1, LEFT
this.of_Register(lv_1, RIGHT)
this.of_SetBarColor(RGB(192, 192, 192))
```

Setting splitbar options

The PFC splitbar control provides options that you can set to control splitbar display. You can control:

- **Bar color** To specify bar color, call the of_SetBarColor function:

  ```
  this.of_SetBarColor(RGB(192,192,192))
  ```

- **The bar color that displays when the bar is moved** To specify bar move color, call the of_SetBarMoveColor function:

  ```
  this.of_SetBarMoveColor(RGB(128, 128, 128))
  ```

- **Horizontal pointer** To specify the name of the pointer that displays when the cursor is over a horizontal splitbar, call the of_SetHorizontalPointer function:

  ```
  this.of_SetHorizontalPointer("SizeNS!")
  ```

- **Vertical pointer** To specify the name of the pointer that displays when the cursor is over a vertical splitbar, call the of_SetVerticalPointer function:

  ```
  this.of_SetVerticalPointer("SizeNS!")
  ```

- **Minimum object size** To specify the minimum size for objects resized by the splitbar, call the of_SetMinObjectSize function:

  ```
  this.of_SetMinObjectSize(100)
  ```

You set these options in the u_st_splitbar instance's Constructor event.

Using the progress bar control

You use u_progressbar (the PFC progress bar control) to provide users with a visual representation of percentage complete for long-running operations. The PFC progress bar can be vertical or horizontal and can display either percent complete or programmatically specified text.

You can display the progress bar control to show percent complete for any repetitive process. Typical uses for a progress bar include:

- Application setup
- Data retrieval
- File copy operations

PFC provides similar progress-bar capabilities on an MDI frame with MicroHelp as part of n_cst_winsrv_statusbar (the window status-bar service).

Other PFC progress bar controls
PFC also provides horizontal (u_hpb) and vertical (u_vpb) progress bars that are based on the standard PowerBuilder progress bar controls. The u_progressbar control has functionality that is not available with the standard controls.

How PFC calculates percent complete

You call the of_SetMaximum function to specify the value that must be reached to equal 100%—for example, the number of rows to be retrieved from the database. Then your code updates the current progress by calling the of_Increment function regularly—for example, once for every ten rows. Percent complete is equal to (current progress / maximum) * 100—for example, (number of rows retrieved / maximum number of rows) * 100.

Showing progress for retrieval
To show progress for row retrieval, code an embedded SQL statement (SELECT MAX) to determine the number of rows to be retrieved. Then increment progress in the DataWindow's RetrieveRow event. (Repeated execution of the RetrieveRow event causes poor performance; but some users prefer visual feedback to optimized performance.)

Using the progress bar in a window

You can either place the progress object directly onto a window or create a pop-up window that displays the progress bar, perhaps including a Cancel button.

v **To use the progress bar in a window or user object:**

1. Place an instance of u_progressbar on the window or user object, optionally making it hidden.
2. Add code to the progressbar control to establish default behavior (this example uses the control's Constructor event):

   ```
   this.of_SetFillStyle(LEFTRIGHT)
   this.of_SetDisplayStyle(PCTCOMPLETE)
   this.of_SetFillColor(RGB(128, 128, 128))
   ```

3. Establish the value that must be reached to equal 100%, such as the number of rows displayed in a DataWindow or the number of bytes in the file to be copied (this example sets the maximum as the number of elements in the array containing DataWindow objects to be printed):

   ```
   DataStore lds_data
   Long ll_return
   Integer li_count, li_max
   ```

```
String ls_dataobject[] = &
  {"d_empall", "d_empbydept", "d_dept"}

lds_data = CREATE DataStore
li_max = UpperBound(ls_dataobject)
uo_progress.of_SetMaximum(li_max)
```

4 Before entering the process to be tracked, initialize the progress bar by calling the of_SetPosition function passing zero:

```
uo_progress.of_SetPosition(0)
```

5 At various points in your process (or at regular points in repetitive logic), call the of_Increment function to update the progress bar:

```
FOR li_count = 1 TO li_max
  lds_data.DataObject = ls_dataobject[li_count]
  lds_data.SetTransObject(SQLCA)
  ll_return = lds_data.Retrieve()
  IF ll_return <> -1 THEN
    uo_progress.of_Increment(1)
    lds_data.Print()
  END IF
NEXT
DESTROY lds_data
```

Using the progress bar in the status bar

N_cst_winsrv_statusbar (the PFC status bar object) includes much of the functionality provided by u_progressbar. This allows you to display progress in the status bar instead of using space in the window or opening a separate progress window.

Status bar service functions that apply to the progress bar are equivalent to progress bar control functions except that the function names include the word *bar*. For example, of_SetBarAutoReset is equivalent to of_SetAutoReset.

v **To use the progress bar in the status bar:**

1 Ensure that the status bar service is enabled for the frame.

2 Enable the progress bar by calling the n_cst_winsrv_statusbar of_SetBar function:

```
this.inv_statusbar.of_SetBar(TRUE)
```

3 Add code to establish default behavior for the progress bar:

```
this.inv_statusbar.of_SetBarDisplayStyle &
   (this.inv_statusbar.PCTCOMPLETE)
```

4 Establish the value that must be reached to equal 100%, such as the number of rows displayed in a DataWindow or the number of bytes in the file to be copied. This example sets the maximum as the number of elements in the array containing DataWindow objects to be printed:

```
DataStore lds_data
Long ll_return
Integer li_count, li_max
String ls_dataobject[] = &
  {"d_empall", "d_empbydept", "d_dept"}

lds_data = CREATE DataStore
li_max = UpperBound(ls_dataobject)
this.inv_statusbar.of_SetBarMaximum(li_max)
```

5 Before entering the process to be tracked, initialize the progress bar by calling the of_SetBarPosition function, passing zero:

```
w_frame  lw_frame

lw_frame = gnv_app.of_GetFrame()
lw_frame.inv_statusbar.of_SetBarPosition(0)
```

6 At various points in your process (or at regular points in repetitive logic), call the of_BarIncrement function to update the progress bar:

```
FOR li_count = 1 TO li_max
  lds_data.DataObject = ls_dataobject[li_count]
  lds_data.SetTransObject(SQLCA)
  ll_return = lds_data.Retrieve()
  IF ll_return <> -1 THEN
    lw_frame.inv_statusbar.of_BarIncrement(1)
    lds_data.Print()
  END IF
NEXT
```

Progress bar options

The PFC progress bar control provides options that you can set to control progress bar behavior. With the exception of of_Maximum and of_Minimum, you typically call these functions in the Constructor event of the u_progressbar instance (u_progressbar) or the pfc_PreOpen event of the Frame window (n_cst_winsrv_statusbar).

Progress bar options include:

- **Maximum and minimum values** Control the values that determine 0% and 100%. You call different functions, depending on whether the progress bar is in a window or in the status bar:

  - **U_progressbar** Call the of_SetMaximum and of_SetMinimum functions:

    ```
    SELECT COUNT(emp_id)
       INTO :il_max
       FROM Employee
       USING SQLCA;
    IF il_max > 0 THEN
       uo_progress.of_SetMaximum(il_max)
       uo_progress.of_SetMinimum(0)
    END IF
    ```

  - **N_cst_winsrv_statusbar** Call the of_SetBarMaximum and of_SetBarMinimum functions:

    ```
    w_frame   lw_frame

    lw_frame = gnv_app.of_GetFrame()
    SELECT COUNT(emp_id)
      INTO :il_max
      FROM Employee
      USING SQLCA;
    IF il_max > 0 THEN
      lw_frame.inv_statusbar.of_SetBarMaximum  &
        (il_max)
      lw_frame.inv_statusbar.of_SetBarMinimum(0)
    END IF
    ```

- **Display style** Controls the text that displays in the progress bar:

  - No text (bar only)
  - Percent complete
  - Current increment value
  - User-specified text

  You call different functions depending on whether the progress bar is in a window or in the status bar:

  - **U_progressbar** Call the of_SetDisplayStyle function, passing either an integer or a u_progressbar constant to specify the information displayed on the progress bar:

    ```
    this.of_SetDisplayStyle(PCTCOMPLETE)
    ```

    - **N_cst_winsrv_statusbar** Call the of_SetBarDisplayStyle function, passing either an integer or an n_cst_winsrv_statusbar constant to specify the information displayed on the progress bar:

      ```
      this.inv_statusbar.of_SetBarDisplayStyle &
        (this.inv_statusbar.PCTCOMPLETE)
      ```

- **Fill style** Controls whether the progress bar fills from left to right, right to left, bottom to top, or top to bottom. You call different functions depending on whether the progress bar is in a window or in the status bar:

    - **U_progressbar** Call the of_SetFillStyle function, passing either an integer or a u_progressbar constant to specify the progress bar fill style:

      ```
      this.of_SetFillStyle(LEFTRIGHT)
      ```

    - **N_cst_winsrv_statusbar** Call the of_SetBarFillStyle function, passing either an integer or an n_cst_winsrv_statusbar constant to specify the progress bar fill style:

      ```
      this.inv_statusbar.of_SetBarFillStyle &
        (this.inv_statusbar.LEFTRIGHT)
      ```

- **Fill color** Controls the color displayed as the progress bar fills. You call different functions depending on whether the progress bar is in a window or in the status bar:

    - **U_progressbar** Call the of_SetColor function, passing the color used to fill the bar:

      ```
      this.of_SetFillColor(RGB(128, 128, 128))
      ```

    - **N_cst_winsrv_statusbar** Call the of_SetBarColor function, passing the color used to fill the bar:

      ```
      this.inv_statusbar.of_SetBarFillColor &
        (RGB(255, 0, 0))
      ```

- **Background color** Controls the color displayed before the progress bar fills:

    - Call the of_SetBackColor function, passing the background color:

      ```
      this.of_SetBackColor(RGB(128, 128, 128))
      ```

Not used in the status bar
This option is not available when using a progress bar with the status bar service.

- **Text color** Controls the color of text displayed in the progress bar. You call different functions depending on whether the progress bar is in a window or in the status bar:

  - **U_progressbar** Call the of_SetTextColor function, passing the color used for text:

    ```
    this.of_SetTextColor(RGB(255, 0, 0))
    ```

  - **N_cst_winsrv_statusbar** Call the of_SetBarTextColor function, passing the color used for text:

    ```
    this.inv_statusbar.of_SetBarTextColor &
      (RGB(255, 0, 0))
    ```

- **Autoreset** Controls whether a completed progress bar remains filled when it reaches 100%. You call different functions depending on whether the progress bar is in a window or in the status bar:

  - **U_progressbar** Call the of_SetAutoReset function:

    ```
    this.of_SetAutoReset(TRUE)
    ```

  - **n_cst_winsrv_statusbar** Call the of_SetBarAutoClear function:

    ```
    this.inv_statusbar.of_SetBarAutoReset(TRUE)
    ```

- **Default step value** Controls the default increment value. This is the value used when you call of_Increment (or of_BarIncrement) with no arguments. The initial default step value is 10. You call different functions depending on whether the progress bar is in a window or in the status bar:

  - **U_progressbar** Call the of_SetStep function, specifying the increment value to be used when calling of_Increment with no arguments:

    ```
    this.of_SetStep(5)
    ```

  - **N_cst_winsrv_statusbar** Call the of_SetBarStep function, specifying the increment value to be used when calling of_BarIncrement with no arguments:

    ```
    this.inv_statusbar.of_SetBarStep(5)
    ```

- **Font options** Controls the font and other display characteristics for the progress bar text. You can call one or more of the following functions:

  ```
  this.of_SetFontBold(TRUE)
  this.of_SetFontFace("Monotype Corsiva")
  this.of_SetFontFamily(Script!)
  this.of_SetFontCharSet(1)
  this.of_SetFontItalic(TRUE)
  ```

```
this.of_SetFontSize(10)
this.of_SetFontPitch(Variable!)
this.of_SetFontUnderline(TRUE)
```

Not used in the status bar
Font options are not available when using a progress bar with the status bar service.

v **To specify user-defined text displayed in the progress bar:**

1 Call the of_SetMessageText function, specifying the text strings to display at regular intervals in the progress bar:

```
...
String  ls_msgtext[ ] = {"Ten", "Twenty", &
  "Thirty", "Forty", "Fifty", "Sixty", &
    "Seventy", "Eighty", "Ninety", &
      "One Hundred"}
this.of_SetMessageText(ls_msgtext)
```

2 Call the of_SetDisplayStyle function, passing either a 3 or the MSGTEXT constant:

```
this.of_SetDisplayStyle(MSGTEXT)
```

The progress bar displays the text strings at regular intervals as the bar fills.

Not used in the status bar
This option is not available when using a progress bar with the status bar service.

CHAPTER 6 Using PFC Windows and Menus

About this chapter

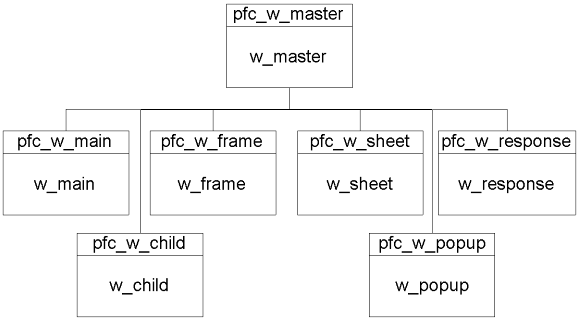
This chapter explains how to use PFC windows and menus.

Contents

Using PFC windows

PFC provides a base class ancestor window (w_master) as well as ancestor windows for each of the standard window types.

Each of these windows descends from w_master:



PFC windows contain instance variables, events, and functions that provide advanced functionality and enable communication with other PFC objects.

Window usage basics

When developing an application, you typically:

- Create base-class and descendent windows
- Enable window services
- Open windows from menu items

Developing with PFC windows

As you begin an application, you review the required functionality to decide which instance variables, events, and functions belong in ancestor windows. The way you define application-specific ancestor behavior differs depending on your extension strategy.

For a discussion of extension strategy, see “Choosing an extension strategy” on page 20.

Enabling window services

PFC provides a variety of window services that you can use to add production-strength features to an application. Many of these services require little or no coding on your part. The window services are:

- Base window service
- Preference service
- Status bar service (frame windows only)
- Sheet management service (frame windows only)
- Resize service (also applies to tabs, tab pages, and custom visual user objects)

v **To use window services:**

1 Determine which window services are appropriate for the window.

2 Enable the appropriate window services using the of_Set*servicename* functions (this example from the window’s pfc_PreOpen event enables the preference and resize services):

```
this.of_SetPreference(TRUE)
this.of_SetResize(TRUE)
```

3 Call other functions as necessary to initialize services (this example enables the preference service for menu items and toolbars and enables the resize service for a DataWindow and two CommandButtons):

```
this.inv_preference.of_SetMenuItems(TRUE)
this.inv_preference.of_SetToolbars(TRUE)
this.inv_resize.of_Register  &
   (dw_emplist, 0, 0, 100, 100)
```

```
this.inv_resize.of_Register  &
   (cb_ok, 0, 100, 0, 0)
this.inv_resize.of_Register  &
   (cb_cancel, 0, 100, 0, 0)
```

For specific usage information on individual window services, see “Window services” on page 89.

Opening PFC windows

In many applications, users open windows by selecting menu items. You can use the PFC message router to help implement this process in a flexible and consistent manner.

See “The message router” on page 41.

v **To open PFC windows from a menu item:**

1 In the menu item, populate Message.StringParm with the window name and call the menu service of_SendMessage function:

```
n_cst_menu  lnv_menu

Message.StringParm = "w_emplist"
lnv_menu.of_SendMessage(this, "pfc_Open")
```

2 In the frame window’s pfc_Open event, add code to access the message object and open the requested window:

```
String   ls_window
w_sheet  lw_sheet

ls_window = Message.StringParm
OpenSheet(lw_sheet, ls_window, this, 4, Original!)
```

Other options
There are other ways to open PFC windows. These include opening windows directly from the menu item, extending the message object to contain passed arguments, and defining additional frame window user events for opening windows.

Using response windows

You typically use w_response (the PFC response window) to create a response window that displays and collects data, settings, or preferences.

Another use for w_response
You can also use w_response to create a response window used in place of a MessageBox. But in that case it's usually best to use the w_message dialog box (part of the error message service).

W_response includes three user events to which you add code that processes the user action:

Event	Use it to	More information
pfc_Apply	Process the window contents, leaving the window open	Many current applications contain an Apply CommandButton that performs this functionality
pfc_Cancel	Ignore the window contents and close the window	You call this event from the CommandButton to which you assign the Cancel property
pfc_Default	Process the window contents and close the window	You call this event from the CommandButton to which you assign the Default property

v **To use w_response events:**

1 Create a window that descends from w_response.

2 Add controls to handle display and user input.

Use the PFC standard visual user objects
You can use PowerBuilder window controls with PFC. But it's best to use controls that descend from PFC standard visual user objects (u_dw, u_lb, u_sle, u_cb, and so on).

3 Add code to support these controls. For example, add code to the window's pfc_PreOpen event to access values in an INI file for display in SingleLineEdit controls.

4 Code the pfc_Apply, pfc_Cancel, and pfc_Default events as necessary. For example, a pfc_Default event might save the window contents in an INI file and close the window:

```
String   ls_temp

ls_temp = trim(sle_base.Text)
SetProfileString("eisapp.ini",  &
   "Files", "base", ls_temp)
ls_temp = trim(sle_x1.Text)
SetProfileString("eisapp.ini",  &
   "Files", "extra1", ls_temp)
ls_temp = trim(sle_x2.Text)
SetProfileString("eisapp.ini",  &
   "Files", "extra2", ls_temp)
Close(this)
```

5 Add CommandButtons to trigger the corresponding event. For example, an OK CommandButton should call the pfc_Default event:

```
parent.Event pfc_Default()
```

Using pfc_Apply
To maximize reusability, place processing in pfc_Apply and call pfc_Apply from pfc_Default.

Using the pfc_Save process

The w_master pfc_Save event automatically validates and saves changes for all PFC and non-PFC DataWindows on a window. Because pfc_Save calls many other events, offloading most of the work to the logical unit of work service, you should think of it as a process rather than a single event.

The logical unit of work service
The pfc_Save process uses the logical unit of work service, which you enable by calling the of_SetLogicalUnitOfWork function. If you do not enable the logical unit of work service, w_master enables it automatically as needed.

Although there are many ways to save data, it’s best to use the pfc_Save event to save changes. In addition to simply calling the w_master pfc_Save event and checking the return value, you can have complete control over update processing by customizing and extending events called by the pfc_Save process. You can:

- Save changes for other self-updating objects (including the n_ds DataStore, the u_tvs TreeView, and the u_lvs Listview)
- Save changes for other controls
- Control which objects are updated and the order in which they are updated
- Save changes for objects on other windows

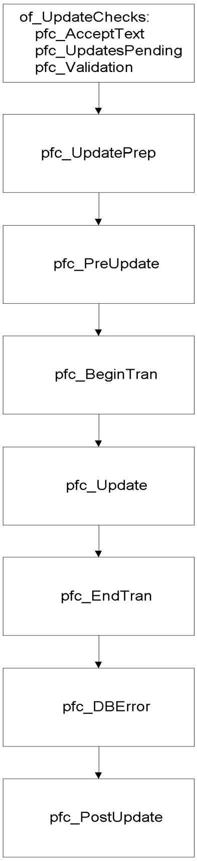
Self-updating objects

PFC integrates update functionality into certain objects called **self-updating objects**. When you call the w_master pfc_Save event, it automatically updates all self-updating objects on the window. All DataWindows are self-updating. You must explicitly enable self-updating functionality for n_ds, u_lvs, u_tab, and u_tvs. And you can add self-updating functionality to any visual or nonvisual control.

For more on self-updating objects, see “Logical unit of work service” on page 111.

The pfc_Save process

This is the w_master pfc_Save process:



Pfc_Save events

These are the events in the pfc_Save process:

Event	Purpose	Comment
pfc_AcceptText	Performs an AcceptText function for all self-updating objects on the window	Called by the w_master of_AcceptText function
pfc_UpdatesPending	Determines which self-updating object have pending updates	Called by the w_master of_UpdatesPending function

Event	Purpose	Comment
pfc_Validation	Performs validation on all self-updating objects with pending updates	Called by the of_Validation function For non-PFC DataWindows, code a ue_validation user event that returns an integer or long greater than or equal to 0 to indicate success
pfc_UpdatePrep	Empty user event to which you add optional update preparation logic	Extend this event if the window itself functions as a self-updating object
pfc_PreUpdate	Empty event in which you can code additional validation	Return 1 for success; return anything else to terminate the pfc_Save process
pfc_BeginTran	Empty event in which you code logic to begin the database transaction, if required by your DBMS	Return 1 for success; return anything else to terminate the pfc_Save process
pfc_Update	Performs database updates for all modified self-updating object	You can extend this event to update controls that are not self-updating. Return 1 for success; return -1 for failure. If you return -1, also create an error message by calling the of_SetDBErrorMsg function. The pfc_DBError event will display this message
pfc_EndTran	Empty event to which you code logic to commit or roll back the database transaction	Commit or roll back changes based on the passed argument Although you can code COMMIT and ROLLBACK statements in other places, it's best to code them here

Event	Purpose	Comment
pfc_DBError	If any update failed, this event displays a message	If the Error service is enabled, this event calls the of_Message function; if not, it calls the PowerScript MessageBox function PFC delays displaying the error message (as opposed to displaying it in pfc_Update) so that you can roll back changes in the pfc_EndTran event before displaying an error message dialog box
pfc_PostUpdate	Resets the update flags for all updated objects, as follows	If you extended the pfc_Save process to update other controls, extend this event to reset the update flags

Pfc_Save return values

Pfc_Save returns values as follows:

Return value	Meaning	Comment
1	Success	
0	No pending updates	
-1	AcceptText error	pfc_Save process halted
-2	Error in pfc_UpdatesPending	pfc_Save process halted
-3	Validation error	pfc_Save process halted
-4	Error in pfc_PreUpdate	pfc_Save process halted
-5	Error in pfc_BeginTran	pfc_Save process halted
-6	Error in pfc_Update	pfc_EndTran and pfc_DBError events completed; pfc_PostUpdate was not performed
-7	Error in pfc_EndTran	pfc_PostUpdate was not performed
-8	Error in pfc_PostUpdate	
-9	Error in pfc_UpdatePrep	pfc_Save process halted

Adding code to extend pfc_Save events

You can customize and extend the pfc_Save process. For example, you can:

- Add code to the pfc_EndTran event to commit and roll back transactions
- Create and code a ue_Validation user event in non-PFC DataWindows to perform validation
- Extend the pfc_Save process to include other types of window controls

v **To add code to the pfc_EndTran event:**

- Add code to the pfc_EndTran event that checks the ai_update_results argument and commits or rolls back changes as appropriate for each transaction object that might be updated:

```
Integer  li_return

IF ai_update_results = 1 THEN
  li_return = SQLCA.of_Commit()
ELSE
  li_return = SQLCA.of_Rollback()
END IF

IF li_return = 0 THEN
  Return 1
ELSE
  Return -1
END IF
```

Enabling self-updating objects

By default, DataWindows are the only self-updating objects that are updatable. All others (n_ds, u_lvs, u_tvs, u_tab, and any user-defined custom visual user objects) are nonupdatable and must be specifically enabled.

v **To enable self-updating objects:**

1 Call the self-updating object’s of_SetUpdatable function:

```
ids_data.of_SetUpdateable(TRUE)
lv_1.of_SetUpdateable(TRUE)
tv_1.of_SetUpdateable)(TRUE)
```

2 (DataStores only) Add the n_ds-based DataStore to the list of controls to be updated:

```
PowerObject lpo_objs[ ]
Integer  li_count

// this = window
lpo_objs = this.control
li_count = UpperBound(lpo_objs)
lpo_objs[li_count + 1] = ids_data
this.of_SetUpdateObjects(lpo_objs)
```

Enabling a one-time update

PFC allows you to identify a specified group of controls and update them.

v **To perform a one-time update:**

1 Identify the controls to be updated:

```
PowerObject lpo_objs[ ]
Integer  li_return

lpo_objs[1] = lv_1
lpo_objs[2] = dw_1
lv_1.of_SetUpdateable(TRUE)
```

2 Perform the one-time save by calling the pfc_SaveObjects event:

```
// this = window
li_return = this.Event pfc_SaveObjects(lpo_objs)
// Check for return codes 1 to -8
...
```

Using menus with PFC

PFC implements menu services through functions, menu items, and events coded in m_master. M_master contains:

- A function that calls the message router
- Menu items that use the message router to perform the requested functionality

Inherit from menus in the extension level
When using menus, always inherit from menus with the m_ prefix (don’t inherit from menus with the pfc_ prefix). Pfc_ prefixed objects are subject to change when you upgrade PFC versions.

Two menu inheritance strategies

You can use PFC's menus or write your own.

Using PFC menus

Your application can use PFC menus as the basis for its menus. In most cases, you use m_master as the ancestor for your application's sheet windows and use m_frame as the frame menu. You add all application-specific menu bar items and menu items to m_master. Let sheet menus inherit from m_master, and enable and disable menu items as appropriate. Disable menu bar items and menu items as appropriate in m_frame.

Creating your own menus

Alternatively, you can implement your own customized menus, separate from m_master. If you do, consider using the menu service of_SendMessage menu function to implement PFC message router functionality.

Extending PFC menus

If you use PFC menus, you will need to modify them or their descendants to provide application-specific processing. When you add new menu bar items and menu items, PFC uses the PowerBuilder Shift Over/Down attribute to control where menu items are placed:

On this menu	PowerBuilder inserts new items
Menu bar	Between Tools and Window
File menu	Above Delete
Edit menu	Above Update Links
View menu	Above Ruler
Tools menu	Above Customize Toolbars
Window menu	Below Undo
Help menu	Above About

For complete information on the Shift Over/Down feature, see the *PowerBuilder User's Guide*.

Creating your own menus

PFC menus provide menu items to cover most events in PFC controls. Your application may have more specific requirements that justify creating a menu from scratch for use with your PFC windows.

Creating menus

Use the Menu painter to create your menus. Add just the items required for your application, defining shortcut keys, accelerators, and toolbar bitmaps as needed.

Creating an extension level
If you are an object administrator creating menus for use by multiple developers and applications, consider creating an ancestor menu (with PowerScript and PFC code) and an empty extension level menu for use by developers.

Communicating with windows

Depending on your needs, you can use either your own menu-window communication method or the PFC message router. When using the message router, it's best to use the menu service of_SendMessage function to call events on the window. Each menu item calls of_SendMessage, passing the name of the event to call. For example, the Clicked event for the Edit>Cut menu item calls of_SendMessage as follows:

```
n_cst_menu  lnv_menu

lnv_menu.of_SendMessage(this, "pfc_Cut")
```

There are two menu items that require special attention:

- **File>Exit** Call the application manager pfc_Exit event:

  ```
  gnv_app.Event pfc_Exit()
  ```

- **MRU menu items (File menu)** Copy the menu item text to Message.StringParm before calling of_SendMessage:

  ```
  n_cst_menu  lnv_menu

  Message.StringParm = this.Text
  lnv_menu.of_SendMessage(this, "pfc_MRUClicked")
  ```

Copy and paste
You can save time by copying and pasting menu item scripts from pfc_m_master.

Enabling items on the Window menu

When the window sheet manager service is enabled, PFC menus automatically enable and disable Window menu items as appropriate. If you are using the sheet manager service and want that functionality in your menus, copy the code from the pfc_m_master Window menu item Selected event.

Using standard menu items

M_master contains menu items that invoke user events on the corresponding window. Use menu items as follows:

- **If the menu item does not apply to a window** Make it invisible.
- **If the menu item applies to a window** Review PowerScript code in the corresponding user event for the associated window, DataWindow, or visual control.

Each of the m_master menu items triggers certain events. Some of these user events are empty; you must add the appropriate PowerScript code to perform application-specific processing.

For more information on PFC user events, see the *PFC Object Reference*.

File menu

Menu item	Event triggered	Object(s) containing user event
New	pfc_New	w_master
Open	pfc_Open	u_rte, w_master
Close	pfc_Close	w_master
Save	pfc_Save	u_rte, w_master
Save As	pfc_SaveAs	u_rte, w_master
Print	pfc_Print	u_dw, u_rte, w_master
Print Preview	pfc_PrintPreview	u_dw, u_rte
Page Setup	pfc_PageSetup	u_dw, w_master
Print Immediate	pfc_PrintImmediate	u_dw, u_rte, w_master
Delete	Empty menu item	Add your own events or functions
Properties	Empty menu item	Add your own events or functions
Exit	pfc_exit	N_cst_appmanager

Edit menu

Menu item	Event triggered	Object(s) containing user event
Undo	pfc_Undo	U_dw, u_em, u_mle, u_rte, and u_sle
Cut	pfc_Cut	U_ddlb, u_ddplb, u_dw, u_em, u_mle, u_oc, u_rte, and u_sle
Copy	pfc_Copy	U_ddlb, u_ddplb, u_dw, u_em, u_mle, u_oc, u_rte, and u_sle
Paste	pfc_Paste	U_ddlb, u_ddplb, u_dw, u_em, u_mle, u_oc, u_rte, and u_sle
Paste Special	pfc_PasteSpecial	U_oc
Clear	pfc_Clear	U_ddlb, u_ddplb, u_dw, u_em, u_mle, u_oc, u_rte, and u_sle

Menu item	Event triggered	Object(s) containing user event
Select All	pfc_SelectAll	U_ddlb, u_ddplb, u_dw, u_em, u_mle, u_rte, and u_sle
Find	pfc_FindDlg	U_dw and u_rte
Replace	pfc_ReplaceDlg	U_dw and u_rte
Update Links	pfc_UpdateLinks	U_oc
Object>Edit	pfc_EditObject	U_oc
Object>Open	pfc_OpenObject	U_oc

View menu

Menu item	Action	Object containing user event
Ruler	pfc_Ruler	U_dw and u_rte
Large Icons	Empty menu item.	Add logic to u_lvs to switch to large icon view
Small Icons	Empty menu item	Add logic to u_lvs to switch to small icon view; call from this menu item
List	Empty menu item	Add logic to u_lvs to switch to list view; call from this menu item
Details	Empty menu item	Add logic to u_lvs to switch to detail view; call from this menu item
Arrange Icons>By	Empty menu item	Add logic to u_lvs to arrange icons by some common property; call from this menu item
Arrange Icons>Auto Arrange	Empty menu item	Add logic to u_lvs to arrange icons; call from this menu item
First	pfc_FirstPage	U_dw and u_rte
Next	pfc_NextPage	U_dw and u_rte
Prior	pfc_PreviousPage	U_dw and u_rte
Last	pfc_LastPage	U_dw and u_rte
Sort	pfc_SortDlg	U_dw
Filter	pfc_FilterDlg	U_dw
Zoom	pfc_Zoom	U_dw

Insert menu

Menu item	Event triggered	Object(s) containing user event
File	pfc_InsertFile	U_rte
Picture	pfc_InsertPicture	U_rte
Object	pfc_InsertObject	U_oc

Tools menu

Menu item	Event triggered	Object(s) containing user event
Customize Toolbars	pfc_Toolbars	W_frame

Window menu

Menu item	Action	Object containing user event
Cascade	pfc_Cascade	W_frame
Tile Horizontal	pfc_TileHorizontal	W_frame
Tile Vertical	pfc_TileVertical	W_frame
Layer	pfc_Layer	W_frame
Minimize All Windows	pfc_MinimizeAll	W_frame
Undo	pfc_UndoArrange	W_frame

Help menu

Menu item	Event triggered	Object(s) containing user event
Help Topics	pfc_Help	W_master
About	of_About	N_cst_appmanager

Using pop-up menus

PFC also provides pop-up menus for use by your applications and PFC services. The pop-up menu provided depends on the control that you right-click. You can disable the pop-up menu functionality by setting the ib_rmbmenu instance variable to FALSE in a control's Constructor event. You can also extend the pop-up menus to add application-specific functionality.

For a list of the pop-up menus provided for standard controls, see "Using right-mouse button support" on page 131.

CHAPTER 7 PFC Utilities

About this chapter

This chapter describes the PFC utilities and how to use them.

Contents

DataWindow Properties window

The DataWindow Properties window allows you to:

- Selectively enable and disable DataWindow services
- View the PFC syntax for the selected service
- Access and modify DataWindow properties interactively, including:

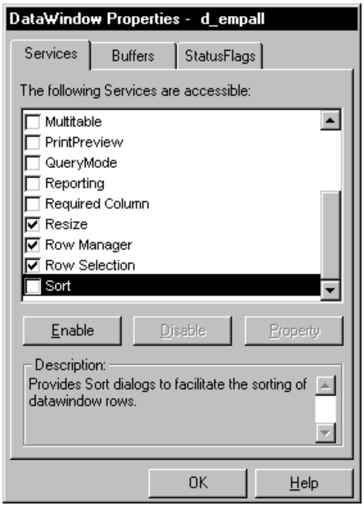
  DataWindow buffers
  Row and column status
  Statistics
  Properties of all objects on the DataWindow object

The DataWindow Properties service is invoked in Lesson 4, “Build the First Sheet Window” of the PFC tutorial.

DataWindow Properties window tab

The DataWindow Properties window has three tabs:

- **Services** Displays a list of DataWindow services. Select a service and click Enable or Disable as needed. Click Properties to display information for the currently selected service:

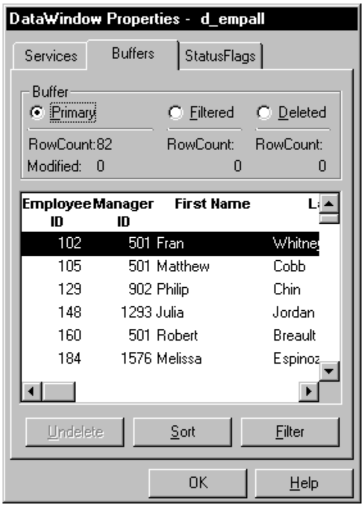


The following Services are accessible:

□ Multitable
□ PrintPreview
□ QueryMode
□ Reporting
□ Required Column
☑ Resize
☑ Row Manager
☑ Row Selection
□ Sort

Description:
Provides Sort dialogs to facilitate the sorting of datawindow rows.

- **Buffers** Click the right mouse button to display a pop-up menu that allows you to manipulate rows:

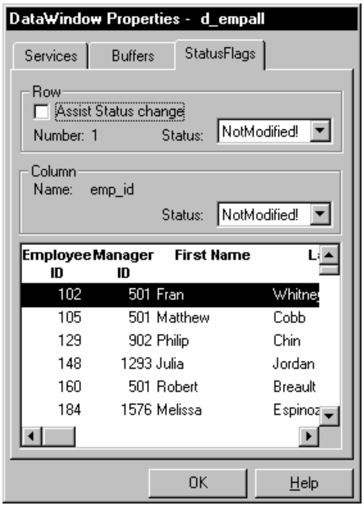


Buffer
● Primary ○ Filtered ○ Deleted

Primary	Filtered	Deleted
RowCount:82	RowCount:	RowCount:
Modified: 0	0	0

Employee ID	Manager ID	First Name	L
102	501	Fran	Whitne
105	501	Matthew	Cobb
129	902	Philip	Chin
148	1293	Julia	Jordan
160	501	Robert	Breault
184	1576	Melissa	Espinoz

- **Status flags** Change DataWindow status flags as necessary. The Assist Status Change check box allows you to perform two-step status changes in one step:



☐ Assist Status change

Number: 1 Status: NotModified!

Name: emp_id

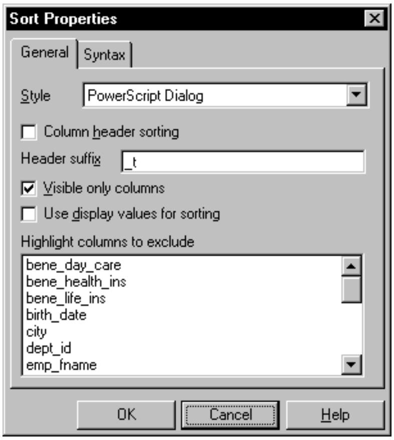
Status: NotModified!

Employee ID	Manager ID	First Name	L
102	501	Fran	Whitne
105	501	Matthew	Cobb
129	902	Philip	Chin
148	1293	Julia	Jordan
160	501	Robert	Breault
184	1576	Melissa	Espinoz

Service dialog box tabs

Each service displays its own set of tabs that display its properties. This example shows the tabs for the sort service:

- **General** Displays information about the selected service:



Style: PowerScript Dialog

☐ Column header sorting

Header suffix: _t

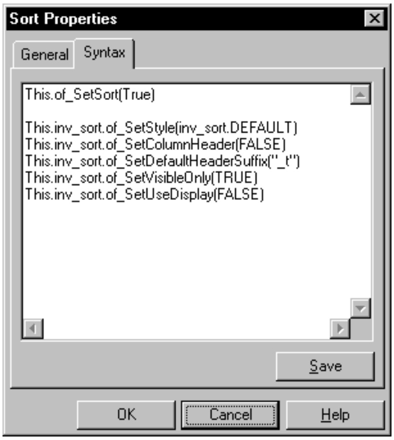
☑ Visible only columns

☐ Use display values for sorting

Highlight columns to exclude

bene_day_care
bene_health_ins
bene_life_ins
birth_date
city
dept_id
emp_fname

- **Syntax** Displays the PFC syntax used for the selected service:



```
This.of_SetSort(True)

This.inv_sort.of_SetStyle(inv_sort.DEFAULT)
This.inv_sort.of_SetColumnHeader(FALSE)
This.inv_sort.of_SetDefaultHeaderSuffix("_t")
This.inv_sort.of_SetVisibleOnly(TRUE)
This.inv_sort.of_SetUseDisplay(FALSE)
```

Usage

Use the DataWindow Properties window to debug and test your application and its use of DataWindow services.

v **To display the DataWindow properties window:**

1 Enable the DataWindow Properties service by calling the u_dw of_SetProperty function:

```
this.of_SetProperty(TRUE)
```

2 When the DataWindow displays, right-click and select DataWindow Properties.

The DataWindow Properties window displays.

SQL Spy

The SQL Spy utility traps and saves SQL automatically for DataWindows and EXEC IMMEDIATE SQL statements. You can also use SQL Spy to display and optionally modify DataWindow SQL statements, and to log native SQL.

Modifying SQL
If you are using an ODBC data source, you must set DisableBind to 1 in the connect string.

SQL Spy logs SQL to a file, which you can optionally display in a pop-up window.

Usage

To use SQL Spy, you call functions to enable the utility, to specify the log file, and to control display of the w_sqlspy window. You can also call a function to log native SQL.

You typically initialize SQL Spy from the application manager pfc_Open event, although you can call SQL Spy functions from anywhere within an application.

v **To enable SQL Spy:**

1 Enable the debugging service by calling the n_cst_appmanager of_SetDebug function:

```
this.of_SetDebug(TRUE)
```

2 Enable SQL Spy by calling the n_cst_debug of_SetSQLSpy function:

```
this.inv_debug.of_SetSQLSpy(TRUE)
```

3 (Optional) Specify a log file by calling the n_cst_sqlspy of_SetLogFile function:

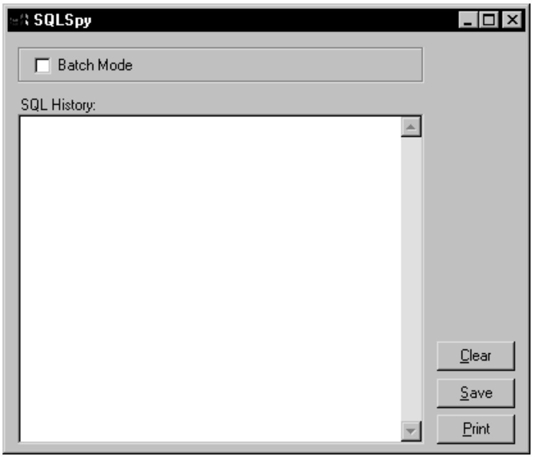
```
this.inv_debug.inv_sqlspy.of_SetLogFile &
       ("c:\MyPFCApps\ThisApp\appdbug.log")
```

v **To display the w_sqlspy pop-up window:**

- Call the n_cst_sqlspy of_OpenSQLSpy function:

```
gnv_app.inv_debug.inv_sqlspy.of_OpenSQLSpy(TRUE)
```

The w_sqlspy pop-up window displays the most recent entries in the log file:



The w_sqlspyinspect dialog box allows you view and optionally modify SQL before the DataWindow submits it to the database.

v **To display the w_sqlspyinspect dialog box:**

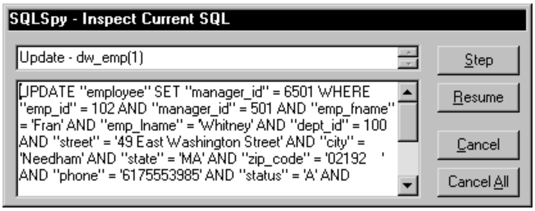
- Call the n_cst_sqlspy of_SetBatchMode(FALSE) function:

```
gnv_app.inv_debug.inv_sqlspy.of_SetBatchMode &
      (FALSE)
```

v **To use the w_sqlspyinspect dialog box:**

1 Update the database, inserting, deleting, or modifying rows.

The w_sqlinspect dialog box displays:



Update - dw_emp(1)

```
UPDATE "employee" SET "manager_id" = 6501 WHERE
"emp_id" = 102 AND "manager_id" = 501 AND "emp_fname"
= 'Fran' AND "emp_lname" = 'Whitney' AND "dept_id" = 100
AND "street" = '49 East Washington Street' AND "city" =
'Needham' AND "state" = 'MA' AND "zip_code" = '02192    '
AND "phone" = '6175553985' AND "status" = 'A' AND
```

2 Review the SQL statement, optionally modifying values.

3 Click the appropriate command button:

- **Step** Updates the current row and displays information for the next row to be updated
- **Resume** Updates the current row and updates all remaining rows; disable the SQL Spy inspect capability
- **Cancel** Does not update the current row and displays the next row to be updated
- **Cancel All** Does not update all remaining rows

v **To log a SQL statement manually:**

- Code the n_cst_sqlspy of_SQLSyntax function, passing the SQL to be logged:

```
String ls_sql

ls_sql = "SELECT * FROM employee;"
gnv_app.inv_debug.inv_sqlspy.of_SQLSyntax &
        ("Native SQL", String(Now()) + ": " + &
            String(Today()) + ": " + ls_sql)
...
```

Security

PFC provides a database-driven security system that requires minimal coding in your applications. It allows you to populate a security database with information, including:

- Window controls
- DataWindow columns
- User objects
- Menu items

You then create a matrix of users and groups, controlling access to these items.

At execution time, PFC selectively enables, disables, or hides secured items, as specified in the security database.

The PFC security system includes:

- **The security administration utility** Allows you to define users, groups, items to be secured, and user access.
- **The security scanner** Scans user-specified objects to gather information on all items that can be secured.
- **The security database** Contains information on users, groups, items to be secured, and user access to those secured items.

Delivered as a local database
PFC provides a local database to hold security information. However, when you implement security, you will want to use a server database.

Security by exception

The PFC security capability provides security by exception. By default, the security system uses the object's current settings. This means that PFC modifies settings only where specified explicitly in the security database.

The process

The security administration utility is a PowerBuilder application you run in order to:

- Define users and groups
- Run the security scanner
- Define security for objects and controls
- Associate users and groups with objects and controls

In your applications, you add code to implement security:

In this object	Add this code
Application manager	Call the n_cst_appmanager of_SetSecurity function to enable the security service, n_cst_security
Application manager or frame window	Establish a Transaction object, connect to the security database, and call the n_cst_security of_InitSecurity function The of_InitSecurity function allows you to set a default group for the user. The security system uses this group if there are no other group settings for the user
Windows that require security	Call the of_SetSecurity function in the window Open event

For more information on enabling security in your applications, see "Implementing security in an application" on page 223.

Defining users and groups

Overview

To use PFC security, you must define users and groups. A user can be a member of zero or more groups; user settings always override group settings.

Usage

You use the security administration utility to define users and groups, as well as associate users with groups.

Use an existing target file
If you already have a target file for your security administration utility, add it to your PowerBuilder workspace, make sure it includes all the PFC libraries in its library list, and skip the first four steps in the following procedure.

v **To define users:**

1 Add an Existing Application target to your PowerBuilder workspace (select File>New from the PowerBuilder menu bar, click the Target tab, click the Existing Application target icon, and click OK).

2 On the Choose Library and Application page of the target wizard, select the pfcsecurity_admin application in the PFCSECAD.PBL library in the PFC\Security directory and click Next.

3 On the Set Library Search Path page of the target wizard, add all the PFC libraries to the library list and click Next.

4 On the Specify Target File page of the target wizard, click Finish.

The default target filename has the same name as the security application you selected with a PBT extension.

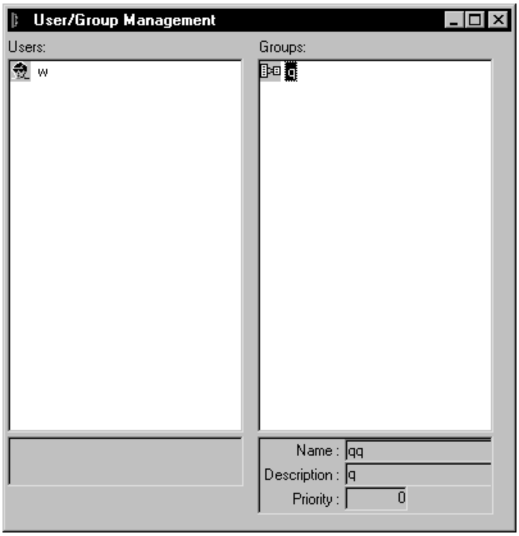
5 Select Run>Select And Run from the PowerBuilder menu bar, select the security_admin target, and click OK.

Security administration database and INI file
If there are problems connecting to the PFC.DB database, check the PFCSECAD.INI file and your workstation’s ODBC settings.

If you have moved the PFC security tables from PFC.DB to some other database, you must update the Database section of the PFCSECAD.INI file to reflect the appropriate database connection parameters.

6 Select File>User/Groups from the security administration utiltiy menu bar.

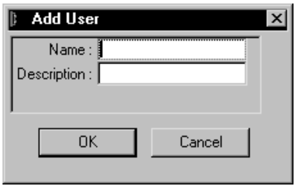
The User/Group Management window displays:



Name : qq
Description : q
Priority : 0

7 Right-click in the Users column and select Add Item.

The Add User dialog box displays:



8 Type a user name and description. The user name must correspond to a user ID that your application can access at runtime.

9 Click OK.

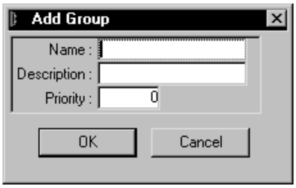
10 Continue adding users as necessary.

11 Select File>Save from the menu bar.

v **To define groups:**

1 Right-click in the Groups column and select Add Group.

The Add Group dialog box displays:



2 Type a group name, description, and priority. Zero is the highest priority; however, user specifications override group specifications.

3 Click OK.

4 Continue adding groups as necessary.

5 Select File>Save from the menu bar.

v **To associate users with groups:**

- Drag the user and drop it over the group.

v **To remove a user from a group:**

1 Right-click on the user and select Delete Item.

The Delete User from Group dialog box displays.

2 Click OK.

3 Select File>Save from the menu bar.

v **To modify a previously defined user or group:**

1 Right-click on the item and select Edit Item.

The Edit User or Edit Group dialog box displays.

2 Modify information as necessary.

3 Click OK.

4 Select File>Save from the menu bar.

Running the security scanner

The security scanner examines all the windows, DataWindows, menus, and user objects in an application. It saves to the PFC database information on:

- Windows
- Window controls
- For DataWindow controls, information on the columns in the associated DataWindow object
- Menu items
- User objects and tab controls; collects information on all controls defined on the user object or tab page

Usage

You can run the security scanner from PowerBuilder or you run it from the security administration utility.

v **To run the security scanner from PowerBuilder:**

1 To your workspace, add an Existing Application target that uses the PFCSECSC.PBL library and the pfcsecurity_scanner application.

Use an existing target if available
If there is already a target file for the pfcsecurity_scanner application, just add this file to your workspace instead of creating a new target file.

2 Select Run>Select And Run, choose the pfcsecurity_scanner target and click OK.

If you have trouble connecting
If there are problems connecting to the PFC.DB database, check the PFCSECAD.INI file and your workstation's ODBC settings.

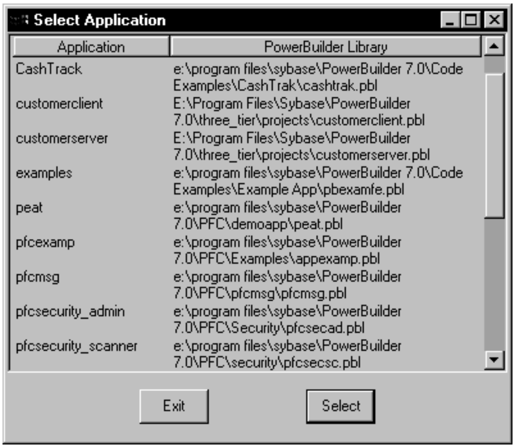
v **To run the security scanner from within the security administration utility:**

1 Select and run the security administration utility (PFCSECAD.PBT).

See "Defining users and groups" on page 215 for steps on adding the security administration target file to your workspace.

2 Select File>Scan Application from the security administration utility menu bar.

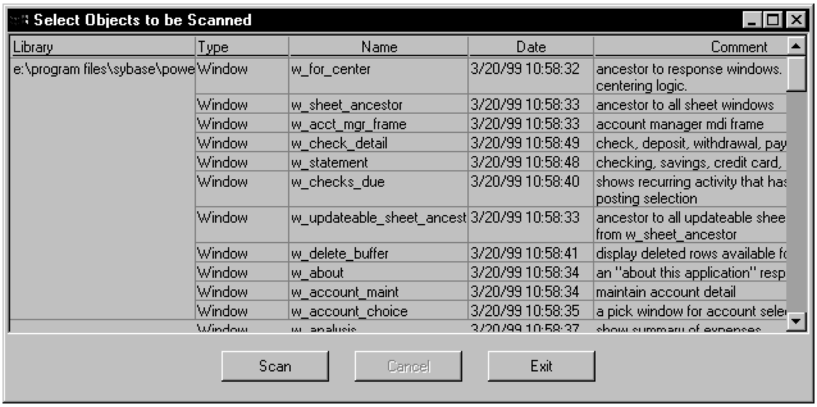
The Select Application dialog box displays a list of the applications defined in the Application section of the PB.INI file:



Application	PowerBuilder Library
CashTrack	e:\program files\sybase\PowerBuilder 7.0\Code Examples\CashTrak\cashtrak.pbl
customerclient	E:\Program Files\Sybase\PowerBuilder 7.0\three_tier\projects\customerclient.pbl
customerserver	E:\Program Files\Sybase\PowerBuilder 7.0\three_tier\projects\customerserver.pbl
examples	e:\program files\sybase\PowerBuilder 7.0\Code Examples\Example App\pbexamfe.pbl
peat	e:\program files\sybase\PowerBuilder 7.0\PFC\demoapp\peat.pbl
pfcexamp	e:\program files\sybase\PowerBuilder 7.0\PFC\Examples\appexamp.pbl
pfcmsg	e:\program files\sybase\PowerBuilder 7.0\PFC\pfcmsg\pfcmsg.pbl
pfcsecurity_admin	e:\program files\sybase\PowerBuilder 7.0\PFC\Security\pfcsecad.pbl
pfcsecurity_scanner	e:\program files\sybase\PowerBuilder 7.0\PFC\security\pfcsecsc.pbl

3 Select the application to be scanned and click Select.

The Select Objects to be Scanned dialog box displays:



Library	Type	Name	Date	Comment
e:\program files\sybase\powe	Window	w_for_center	3/20/99 10:58:32	ancestor to response windows. centering logic.
	Window	w_sheet_ancestor	3/20/99 10:58:33	ancestor to all sheet windows
	Window	w_acct_mgr_frame	3/20/99 10:58:33	account manager mdi frame
	Window	w_check_detail	3/20/99 10:58:49	check, deposit, withdrawal, pay
	Window	w_statement	3/20/99 10:58:48	checking, savings, credit card,
	Window	w_checks_due	3/20/99 10:58:40	shows recurring activity that has posting selection
	Window	w_updateable_sheet_ancest	3/20/99 10:58:33	ancestor to all updateable shee from w_sheet_ancestor
	Window	w_delete_buffer	3/20/99 10:58:41	display deleted rows available fo
	Window	w_about	3/20/99 10:58:34	an "about this application" resp
	Window	w_account_maint	3/20/99 10:58:34	maintain account detail
	Window	w_account_choice	3/20/99 10:58:35	a pick window for account sele
	Window	w_analysis	3/20/99 10:58:37	show summary of expenses

4 Press CTRL+click or SHIFT+click to select the objects to be scanned.

Selecting objects
To minimize the size of the security database, do not select objects for which you will never assign security.

5 Click Scan.

Scanner executable file
The security administration utility uses the *pfcsecsc.exe* file to perform the scan. PowerBuilder installs this file in the PFC Security directory.

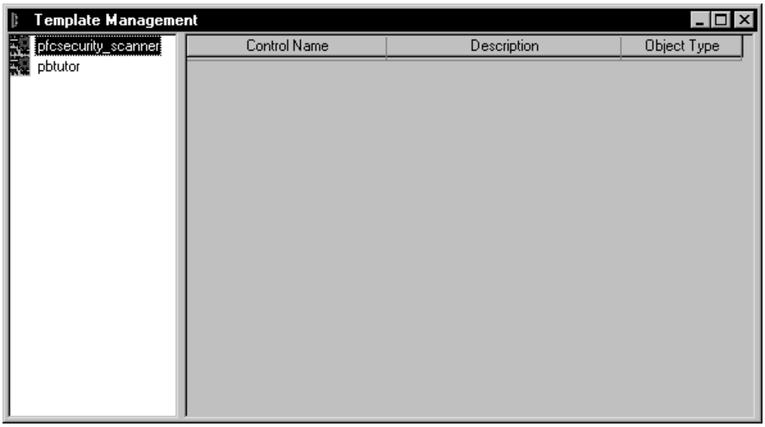
6 When scanning completes, click Exit.

v **To customize the controls for which security is enabled:**

1 Select and run the security administration utility.

2 Select File>Templates from the security administration utility menu bar.

The Template Management window displays:

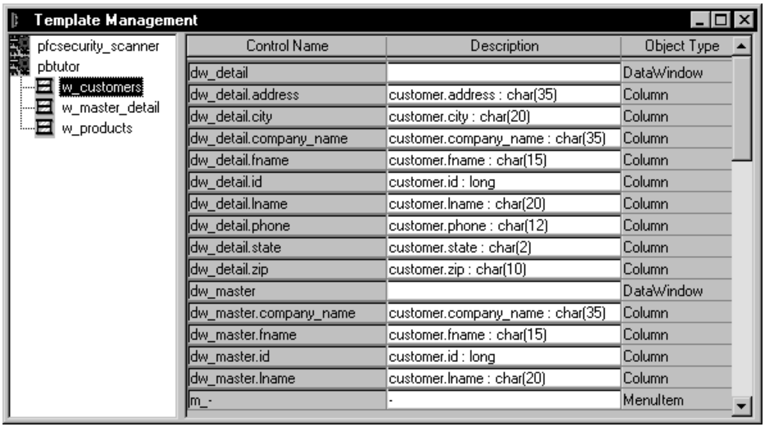


3 Double-click on the application you just scanned.

A list of windows displays.

4 Double-click on one of the windows.

A list of controls displays:



Control Name	Description	Object Type
dw_detail		DataWindow
dw_detail.address	customer.address : char(35)	Column
dw_detail.city	customer.city : char(20)	Column
dw_detail.company_name	customer.company_name : char(35)	Column
dw_detail.fname	customer.fname : char(15)	Column
dw_detail.id	customer.id : long	Column
dw_detail.lname	customer.lname : char(20)	Column
dw_detail.phone	customer.phone : char(12)	Column
dw_detail.state	customer.state : char(2)	Column
dw_detail.zip	customer.zip : char(10)	Column
dw_master		DataWindow
dw_master.company_name	customer.company_name : char(35)	Column
dw_master.fname	customer.fname : char(15)	Column
dw_master.id	customer.id : long	Column
dw_master.lname	customer.lname : char(20)	Column
m_-	-	MenuItem

5 Modify descriptions as appropriate.

Modifying descriptions on this window can make things clearer when associating users and groups with windows, window controls, and menu items.

6 Delete items that you will not secure by right-clicking over the item and selecting Delete.

Deleting unnecessary items reduces the size of the security database, increasing performance.

7 When you are through, select File>Save from the menu bar.

8 Continue this process with all objects to be secured.

Defining security for users and groups

After running the scanner to record objects and controls and selectively deleting items that don’t require securing, you specify security by associating users and groups with objects and controls.

Usage

For each user and group, you enable or disable access to window controls, DataWindow columns, user objects, and menu items for each object to be secured.

Users may belong to zero or more groups. User settings always take precedence over group settings. If there are no user settings, then the group setting with the highest priority is used (zero is the highest priority).

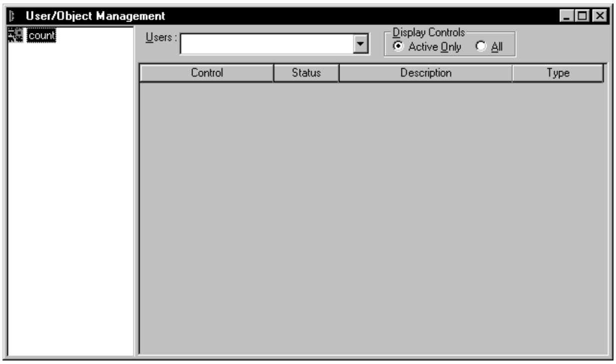
v **To define security for a user or group:**

1 Select and run the security administration utility (PFCSECAD.PBT).

See “Defining users and groups” on page 215 for steps on adding the security administration target file to your workspace.

2 Select File>Users/Objects from the menu bar.

The User/Object Management window displays:



Users :

Display Controls: ● Active Only ○ All

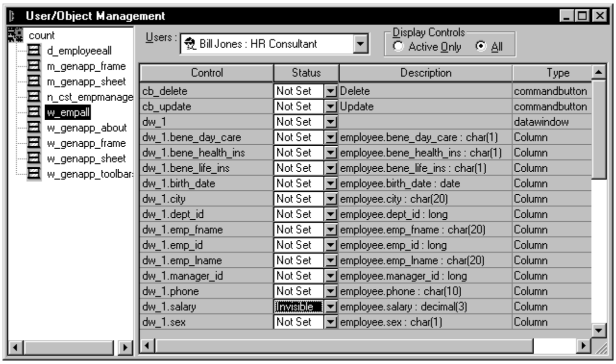
Control	Status	Description	Type

3 Click the Users drop-down list and select a user for whom to set security.

4 Double-click the application containing the objects to be secured.

5 Select the All radio button (if it’s not already selected).

You are now ready to secure items.



Users : Bill Jones : HR Consultant

Display Controls: ○ Active Only ● All

Control	Status	Description	Type
cb_delete	Not Set	Delete	commandbutton
cb_update	Not Set	Update	commandbutton
dw_1	Not Set		datawindow
dw_1.bene_day_care	Not Set	employee.bene_day_care : char(1)	Column
dw_1.bene_health_ins	Not Set	employee.bene_health_ins : char(1)	Column
dw_1.bene_life_ins	Not Set	employee.bene_life_ins : char(1)	Column
dw_1.birth_date	Not Set	employee.birth_date : date	Column
dw_1.city	Not Set	employee.city : char(20)	Column
dw_1.dept_id	Not Set	employee.dept_id : long	Column
dw_1.emp_fname	Not Set	employee.emp_fname : char(20)	Column
dw_1.emp_id	Not Set	employee.emp_id : long	Column
dw_1.emp_lname	Not Set	employee.emp_lname : char(20)	Column
dw_1.manager_id	Not Set	employee.manager_id : long	Column
dw_1.phone	Not Set	employee.phone : char(10)	Column
dw_1.salary	Invisible	employee.salary : decimal(3)	Column
dw_1.sex	Not Set	employee.sex : char(1)	Column

6 For items to be secured, use the Status drop-down list to specify Enabled, Disabled, or Invisible (Not Set makes no change to the object’s settings).

7 When you are finished, select File>Save from the menu bar.

8 Continue with the object until you are finished with all users and groups.

9 Continue with other objects.

Implementing security in an application

Once you have defined a security database, enable the security service in your application.

Usage

Enabling the security service in your application involves:

- Enabling the security service
- Establishing a database connection to the database that contains the security tables and communicating this information to the security service
- Enabling the security service on the appropriate windows

v **To enable the security service for an application:**

- Call the n_cst_appmanager of_SetSecurity function:

```
gnv_app.of_SetSecurity(TRUE)
```

v **To establish a connection to the database containing the security tables and communicate it to the security system:**

1 Create a Transaction object and connect to the database (this example assumes an itr_security instance variable on a customized n_cst_appmanager descendant):

```
gnv_app.itr_security = CREATE n_tr
gnv_app.itr_security.of_Init &
       (gnv_app.of_GetAppINIFile(), "Security")

gnv_app.itr_security.of_Connect()
```

Security table placement
To minimize the number of database connections held by your application, place the security tables in the application database.

2 Call the n_cst_security of_InitSecurity function:

```
Integer li_return

li_return = &
      gnv_app.inv_security.of_InitSecurity &
         (gnv_app.itr_security, "EISAPP", &
            gnv_app.of_GetUserID(), "Default")
```

v **To enable security for a window:**

- Call the n_cst_security of_SetSecurity function in the window's Open or pfc_PreOpen event:

```
gnv_app.inv_security.of_SetSecurity(this)
```

Maintaining the security database

The PFC security system tables are delivered in the PFC.DB local database. The PFC security tables are:

Security_apps
Security_groups
Security_info
Security_template
Security_users

You can use the PFC.DB local database to define users and groups, scan objects, and define access privileges. However, you will need to migrate these tables to a server database before deploying applications. The PFC security system and the security database are designed for easy migration to a server database:

- All database interactions for PFC security are via DataWindows (there is no embedded SQL)
- The PFC security system enforces cascading deletes manually

Usage

To enable security for all of your application's users, move the PFC security tables to a server database.

v **To migrate PFC security tables to a server database:**

1 Use the Pipeline painter to move table definitions and data to a server database. Retain the table and column names as much as possible.

2 Use the DataWindow painter to access the PFC security DataWindows. Note the following:

   - Preserve the DataWindow's column order and DataWindow column names.
   - If necessary, use the Select Painter to change the name of the associated database tables or columns to match those on the server database. Remember not to change the name of the DataWindow columns.

3 In your application, populate Transaction object fields as appropriate for the server database containing the security tables.

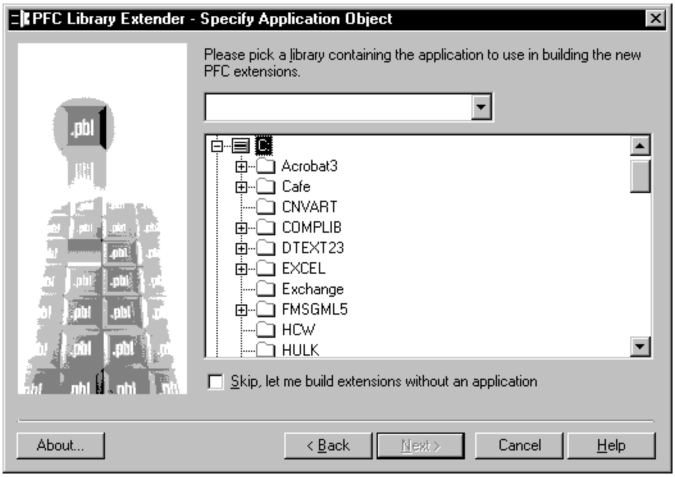
Library Extender

You use the PFC Library Extender to automatically create and populate an intermediate extension level between two existing levels, redefining the inheritance hierarchy.

By supplying the Library Extender with the names of the upper and lower levels, the Library Extender:

- Creates a new PBL
- Creates objects in the new PBL that descend from objects in the upper level

- Recreates objects in the lower level such that they descend from objects in the new PBL



PFC Library Extender - Specify Application Object

Please pick a library containing the application to use in building the new PFC extensions.

□ Skip, let me build extensions without an application

Typically you would use the Library Extender to add an intermediate level (or levels) between the PFC ancestor level and the PFC extension level. Adding corporate and departmental extensions to intermediate extension levels allows the application programmer to make full use of the extension level.

Usage

The Library Extender is available on the Tool tab in the New dialog box.

For complete usage information, see the Library Extender online Help.

Migration Assistant

The Migration Assistant scans PowerBuilder libraries (PBLs) and highlights usage of obsolete functions and events. Obsolete functions and events still work in the current version of PowerBuilder but may not work in future versions. If you plan on maintaining an application in the future, it's best to use current syntax and events.

Usage

The Migration Assistant is available on the Tool tab in the New dialog box.

For complete usage information, see the Migration Assistant online Help.

CHAPTER 8 Deploying a PFC Application

About this chapter

This chapter explains considerations related to PFC application deployment.

Contents

Choosing a deployment strategy

You use PFC to build production-strength applications. As with all production-strength applications, deploying a PFC application requires careful planning and implementation.

Your goal

Your deployment strategy must provide user workstations with everything they need to execute a PFC application:

- Application executable (EXE) file
- Application PBDs or DLLs (if not using a single EXE file)
- PFC PBDs or DLLs (if not using a single EXE file)
- Other files and entries used by your application (ActiveX controls, registry entries, INI files, bitmaps, and so on)
- PowerBuilder execution modules (may already be installed on the user workstation)
- Database client software (may already be installed on the user workstation)

Four deployment options

A PFC application has the same four deployment options as any other PowerBuilder application. Each of these options has relative advantages and disadvantages that you need to consider before choosing a deployment strategy:

Deployment option	Advantages	Disadvantages
Pcode executable	Single file Simple deployment	Large (minimum 3M) To update, you must regenerate the entire application
Compiled executable	Single file Simple deployment Compiled code	Very large (minimum 8M) To update, you must regenerate the entire application
Pcode PBDs	Smaller EXE To update, you can replace a single PBD	Multiple files Needs separate set of physical files
Compiled DLLs	Smaller EXE Compiled code To update, you can replace a single DLL	Multiple files Needs separate set of physical files

Physical file issues If your applications use Pcode PBDs or compiled DLLs, they usually need a separate set of physical files for each deployed PFC application. This is because of the internal interdependencies that trickle down from high-level extension objects, such as w_master, n_cst_dwsrv, n_cst_winsrv, and n_cst_dssrv.

See “Setting up the application manager” on page 29.

Using a common set of physical files
If no deployed application has made changes to either the PFC ancestor layer or the PFC extension layer, applications can share PBD or DLL files. But to ensure ease of maintenance and upgrade, it’s still best to provide separate physical PBD and DLL files for each deployed application.

Using PBR files

PFC ships with six PBR files:

- *One* for use when placing the bitmaps in the *EXE*
- *A set of five* for use when distributing *PBDs or DLLs*

Use these files as follows:

Deployment method	**PBR file(s) to use**	**What happens when you deploy *with* PBR**	**What happens when you deploy *without* PBR**
Placing the bitmaps in a single EXE	Pfc.pbr (This PBR includes dynamic DataWindow references)	Bitmaps and dynamically referenced DataWindow objects are copied into the EXE	*This option will not work* (dynamic DataWindow references will fail)
Distributing PBDs or DLLs	Pfcapsrv.pbr Pfcdwsrv.pbr Pfcmain.pbr Pfcutil.pbr Pfcwnsrv.pbr	The bitmaps named in each PBR file are copied into the associated PBD or DLL	You must deploy bitmap files separately and make sure they are in a directory that is accessible at execution time

Deploying database tables

PFC ships with the PFC.DB database.

Table references

Although no PFC services reference PFC.DB directly, certain services reference tables that were originally shipped in PFC.DB:

PFC service	**Tables referenced**
Error message service (n_cst_error)	Messages
Security service (n_cst_security)	Security_apps Security_groupings Security_info Security_template Security_users

What to do

To minimize the number of database connections held by your application, it's usually best to move these tables to the application database.

In any case, your application deployment strategy must provide for user access to all database tables required by the application. This includes installing client software, updating INI files, updating registry entries, and all other database deployment considerations outlined in the database deployment discussion in *Application Techniques*.

Deploying PFC dialog box Help

PFC includes the PFCDLG.HLP file, which contains online Help for PFC dialog boxes. If your application uses PFC dialog boxes (such as w_find, w_replace, and w_sortdragdrop) you should deploy PFCDLG.HLP so users will have dialog box Help.

PFC also includes PFCDLG.RTF, which contains the source text for PFC dialog box Help. If your application makes specific use of PFC dialog boxes, you can modify this file and recompile the Help file.

PART 4

PFC Tutorial

This part provides a simple tutorial to get you started with PFC.

This part is for all PFC users.

LESSON 1

Generate a PFC Application

In this tutorial, you generate a PFC application using the Template Application wizard. You will then create a user object to inherit from the PFC application manager object and you will redirect events from the Application object to the newly created user object.

In this lesson you will:

- Create a PFC application
- Modify the application manager
- Redefine a global variable and review events
- Use the PFC Transaction Object service

How long will this lesson take?
About 20 minutes.

What will you learn about PFC?

- How to create a PFC application using the Template Application wizard
- How to use n_tr, PFC's customized Transaction object
- How to use the application manager

Create a PFC application

Where you are
> Create a PFC application
Modify the application manager
Redefine a global variable and review events
Use the PFC Transaction Object service

When you start PowerBuilder, you must open an existing workspace or generate a new one. In this exercise, you create an application target in a new PowerBuilder workspace. The application will use and inherit from objects in the PFC libraries.

Required tutorial setup
This tutorial uses the EAS Demo DB database that installs with PowerBuilder. This is an Adaptive Server Anywhere database and requires an Adaptive Server Anywhere engine.

If you do not already have Adaptive Server Anywhere on your local machine or server, you must install it now. (You can install it from the PowerBuilder CD.) If you installed PowerBuilder in a nondefault location, you must make sure that the *odbc.ini* registry entry defining the EAS Demo DB as a data source points to the correct location of the Adaptive Server Anywhere engine.

1 **Select** ***File>New*** **from the PowerBuilder menu bar.**
Make sure that the ***Workspace*** **page of the New dialog box displays.**
Select the ***Workspace*** **wizard and click** ***OK*****.**

A file selection dialog box displays.

2 **Type** ***PFC Tutorial*** **in the File Name box and click** ***OK*****.**

An icon for the new workspace displays in the System Tree. If the System Tree is not displayed, select the System Tree tool in the toolbar or select the System Tree menu item in the Window menu.

3 **Select** ***File>New*** **from the PowerBuilder menu bar.**
Click the ***Target*** **tab of the New dialog box.**
Select the ***Template Application*** **wizard and click** ***OK*****.**

The wizard displays introductory information about itself.

4 **Click *Next* twice until you see the *Specify New Application And Library* page.**
Type *my_pfc_app* in the Application Name box.

The wizard resets default filenames for the application library and target.

5 **Click the *ellipsis* button next to the Library box.**
Navigate to the *PFC tutorial* directory.
Make sure *my_pfc_app.pbl* displays in the File Name box and click *Save*.

In a typical installation, the PFC tutorial directory is: *C:\Program Files\Sybase\PowerBuilder 9.0\PFC\Tutorial*.

6 **Click the *ellipsis* button next to the Target box.**
Navigate to the *PFC tutorial* directory.
Make sure *my_pfc_app.pbt* displays in the File Name box, click *Save* and click *Next*.

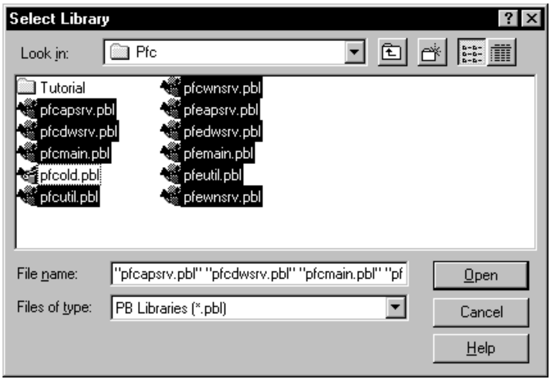
7 **Select the *PFC-Based Application* radio button on the Specify Application Type page and click *Next* again.**

The Adjust Application Library Search Path page displays. The my_pfc_app.pbl file is the only library in the list box.

8 **Click the *ellipsis* button next to the Library Search Path list box.**

A standard library selection dialog box displays.

9 **Navigate to the *PFC* directory (one level above the PFC Tutorial directory).**
Use CTRL+click or SHIFT+click to select these libraries from the main layer and the extension layer:



Look in: Pfc

File name: "pfcapsrv.pbl" "pfcdwsrv.pbl" "pfcmain.pbl" "pf

Files of type: PB Libraries (*.pbl)

PFC main layer libraries	PFC extension layer libraries
pfcapsrv.pbl	*pfeapsrv.pbl*
pfcmain.pbl	*pfemain.pbl*
pfcwnsrv.pbl	*pfewnsrv.pbl*
pfcdwsrv.pbl	*pfedwsrv.pbl*
pfcutil.pbl	*pfeutil.pbl*

10 Click *Open*.

The Library Search Path list box redisplays with PFC libraries added to the list.

11 Click *Next* twice until you see the Ready To Create Application page.

PowerBuilder summarizes your wizard selections in a list box. The Generate To-Do List check box is selected.

12 Click *Finish*.

Modify the application manager

Where you are
Create a PFC application
> Modify the application manager
Redefine a global variable and review events
Use the PFC Transaction Object service

When you use the Template Application wizard to create a PFC application, it redirects Application object processing to a PFC application manager. This strategy provides many benefits, including extensibility and reuse.

You implement the application manager through the n_cst_appmanager custom class user object or a customized descendant. This tutorial implements the application manager by creating a descendant of n_cst_appmanager. You then initialize application-wide variables in the new application manager.

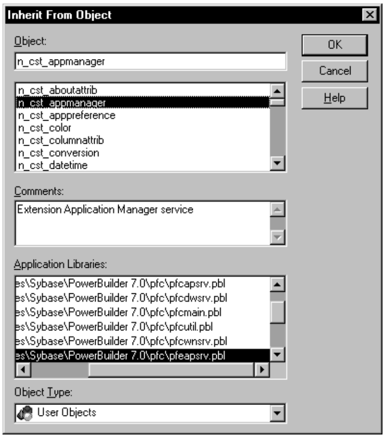
For more information on implementing the application manager, see Chapter 3, “PFC Programming Basics”.

1   **Click the *Inherit* button in the PowerBar.**

The Inherit From Object dialog box displays.

2   **Make sure *My_PFC_App.* displays in the Target list box.**
Select *pfeapsrv.pbl* in the Libraries drop-down list.
Select *User Objects* in the Objects of Type drop-down list.

You can enlarge the dialog box to display the full library name.



Object: n_cst_appmanager

n_cst_aboutattrib
n_cst_appmanager
n_cst_apppreference
n_cst_color
n_cst_columnattrib
n_cst_conversion
n_cst_datetime

Comments: Extension Application Manager service

Application Libraries:
es\Sybase\PowerBuilder 7.0\pfc\pfcapsrv.pbl
es\Sybase\PowerBuilder 7.0\pfc\pfcdwsrv.pbl
es\Sybase\PowerBuilder 7.0\pfc\pfcmain.pbl
es\Sybase\PowerBuilder 7.0\pfc\pfcutil.pbl
es\Sybase\PowerBuilder 7.0\pfc\pfcwnsrv.pbl
es\Sybase\PowerBuilder 7.0\pfc\pfeapsrv.pbl

Object Type: User Objects

3 **Select *n_cst_appmanager* in the Object list box and click *OK*.**

The User Object painter workspace displays. The default view layout scheme for the User Object painter includes a Script view, a Properties view, a Non-Visual Object list, and a stack of tabbed panes. Within the stack you can display the Event List, Function List, or Variable view.

Resetting to the default view layout scheme
To reset the layout to the default scheme, select View>Layouts>Default.

4 **Make sure *Untitled* displays as the name of the user object in the first drop-down list of the Script view.**
Make sure the *Constructor* event displays as the selected event in the second drop-down list of the Script view.

The purple icon in front of the Constructor event name indicates that a script is coded for this event in an ancestor object. In PowerBuilder, events (unlike functions) are triggered first in the ancestor object, then in the descendent object.

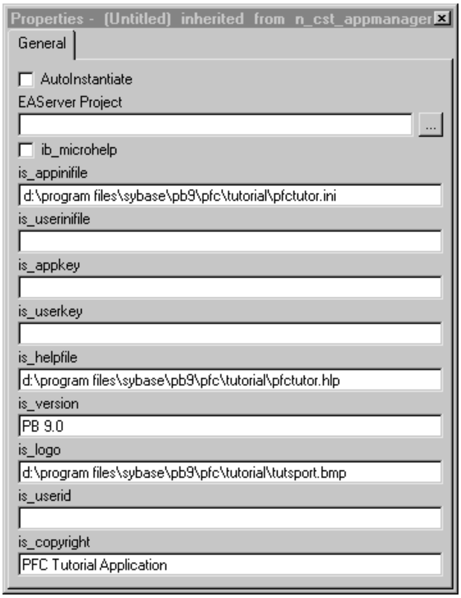
5 **Select *pfc_n_cst_appmanager* from the third drop-down list of the Script view.**
Examine the code for the ancestor object.

Most of the ancestor script is commentary. The code for the ancestor Constructor event consists of an assignment statement (that assigns the Application object to an instance variable) and a call to the GetEnvironment system function that populates the Environment object.

6 **Type the following information in the boxes in the Properties view (replace *drive* and *pathname* with your workstation's path to the PFC directory):**

Property	Value
is_appinifile	*drive*:\pathname\pfc\tutorial\pfctutor.ini
is_helpfile	*drive*:\pathname\pfc\tutorial\pfctutor.hlp
is_version	PFC 9.0
is_logo	*drive*:\pathname\pfc\tutorial\tutsport.bmp
is_copyright	PFC tutorial application

The first two properties establish an INI file and online Help file for use with the application. The other properties establish information that will be used in the application's About dialog box and splash screen.



Properties - (Untitled) inherited from n_cst_appmanager

General

□ AutoInstantiate

EAServer Project

□ ib_microhelp

is_appinifile: d:\program files\sybase\pb9\pfc\tutorial\pfctutor.ini

is_userinifile

is_appkey

is_userkey

is_helpfile: d:\program files\sybase\pb9\pfc\tutorial\pfctutor.hlp

is_version: PB 9.0

is_logo: d:\program files\sybase\pb9\pfc\tutorial\tutsport.bmp

is_userid

is_copyright: PFC Tutorial Application

If you prefer to add code to the Constructor event
You could type the following lines of code calling PFC functions instead of filling in the property boxes in the Properties view:

```
this.of_SetAppIniFile &
("drive:\pathname\pfc\tutorial\pfctutor.ini")
this.of_SetHelpFile &
("drive:\pathname\pfc\tutorial\pfctutor.hlp")
this.of_SetLogo &
("drive:\pathname\pfc\tutorial\tutsport.bmp")
this.of_SetCopyright("PFC Tutorial application.")
this.of_SetVersion("PFC 9.0")
```

7 **Select *File*>*Save As* from the menu bar.**

The Save User Object dialog box displays.

8 Select ***my_pfc_app.pbl*** **in the Application Libraries box.**
Type ***n_cst_tutappmanager*** **in the User Objects box.**
Type the following comment in the Comments box:

```
This is the PFC tutorial application manager.
```

9 Click ***OK*****.**

PowerBuilder saves the user object and redisplays it in the User Object painter workspace. The n_cst_tutappmanager name now displays in the painter title bar, in the Non-Visual Object List view, and in the first drop-down list of the Script view.

10 Select ***File*****>*****Close*** **from the menu bar.**

Redefine a global variable and review events

Where you are
Create a PFC application
Modify the application manager
> Redefine a global variable and review events
Use the PFC Transaction Object service

When you generate a PFC application using the Template Application wizard, all the events of the Application object call corresponding events in the application manager (n_cst_appmanager or a descendant of n_cst_appmanager). For example, the Open event calls the application manager's pfc_Open event, the Close event calls pfc_Close, and so on.

PFC uses the gnv_app global variable to access the application manager at runtime. Now you will modify this variable to use the customized application manager and you will review the events of the Application object that are redirected to the application manager.

1 **Click the *Open* button in the PowerBar.**
Select *my_pfc_app.pbl* in the Libraries list box.
Select *Applications* in the Objects of Type drop-down list.

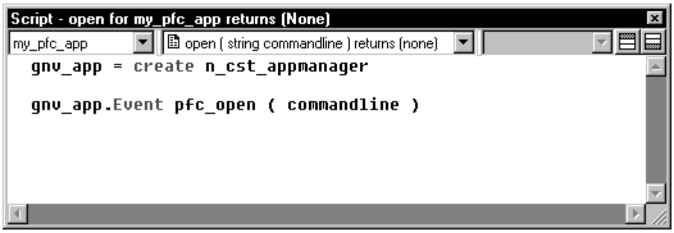
The Object box displays the only application (my_pfc_app) in the library file you selected.

2 **Click *OK*.**

The Application painter displays the my_pfc_app Application object.

3 **Make sure the Script view displays the *Open* event for the *my_pfc_app* Application object.**

The first line of code is an assignment statement that creates an instance of the application manager and assigns it to the global variable gnv_app. The next line of code calls the pfc_Open event of the application manager.


Script - open for my_pfc_app returns (None)
my_pfc_app
open (string commandline) returns (none)

```
gnv_app = create n_cst_appmanager

gnv_app.Event pfc_open ( commandline )
```

4 **Change the first line of code to:**

```
gnv_app = CREATE n_cst_tutappmanager
```

The code will now create an instance of the application manager descendant, not the PFC ancestor. You do not change the second line of code, but you will change the global variable data type declaration.

PFC login window
The PFC login window is not used in this tutorial, but if you want a login window at runtime, you could add the following line to the application object Open event:

```
gnv_app.of_LogonDlg ( )
```

Typing this line before the call to the pfc_open event assures that the login window will open before the connection is made to the database. If you want to make a database connection with user-entered information from the login window, you would have to add code to the Clicked event of the OK button in the w_logon window—and then make sure that that information is not overwritten by information in *pfctutor.ini* that you selected as the application INI file.

5 **Select** ***Global Variables*** **in the second drop-down list in the Variable view.**

The Variable view displays the global variable declarations.

Using the Script view
You can use the Script view instead of the Variable view, if you select Declare in the first drop-down list. PowerBuilder will prevent you from opening two views to the same script, so you won’t be able to do this when the Variable view displays the Global Variables script.

6 **Modify the *gnv_app* global variable declaration to use n_cst_tutappmanager:**

```
n_cst_tutappmanager gnv_app
```

By defining gnv_app as type n_cst_tutappmanager, you gain access to all n_cst_appmanager functionality as well as any new instance variables, user events, and functions defined in n_cst_tutappmanager.

7 **Make sure my_pfc_app displays in the first drop-down list of the Script view.**
Review the Application object's precoded events:

Event	What it does
Close	Calls the application manager's pfc_Close event and destroys gnv_app
ConnectionBegin	Calls the application manager's pfc_ConnectionBegin event, passing three arguments and returning the connection privilege (for use with distributed applications)
ConnectionEnd	Calls the application manager's pfc_ConnectionEnd event (for use with distributed applications)
Idle	Calls the application manager's pfc_Idle event
SystemError	Calls the application manager's pfc_SystemError event

Use the PFC Transaction Object service

Where you are
Create a PFC application
Modify the application manager
Redefine a global variable and review events
> Use the PFC Transaction Object service

Now you will look at the definition for the default Transaction object. In standard PowerBuilder applications, SQLCA (SQL Communications Area) is defined as the default Transaction object. In PFC applications, the default Transaction object is assigned by the PFC transaction object service.

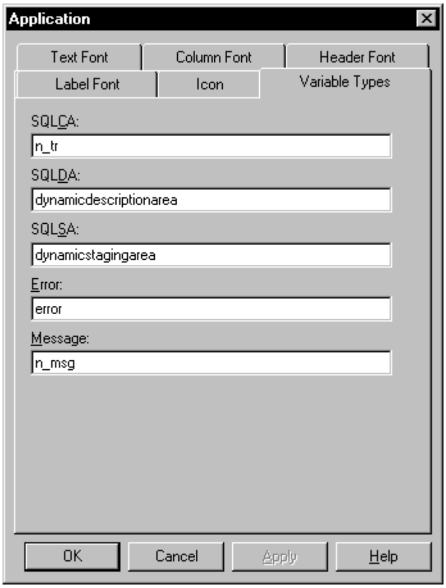
The n_tr user object is a Transaction object defined in *pfemain.pbl*. It inherits from the pfc_n_tr object in the *pfcmain.pbl* library. In this exercise, you will see how the SQLCA Transaction object is registered with the PFC transaction registration service (the registration was set automatically by the Template Application wizard).

If you are not continuing directly from the previous exercise
If you closed the Application painter, you must reopen it to display the global SQLCA variables that are automatically part of any PowerBuilder application.

1 **Click the *Additional Properties* button on the General page of the Properties view for the Application object.**

The Application property sheet shows additional properties in a tab page format.

2 **Click the *Variable Types* tab.**



SQLCA: n_tr

SQLDA: dynamicdescriptionarea

SQLSA: dynamicstagingarea

Error: error

Message: n_msg

3 **Look at the *SQLCA* variable definition.**

PowerBuilder uses PFC's n_tr transaction for SQLCA. If you had generated an application that did not specify PFC libraries, the value assigned to the SQLCA global variable would have been "transaction" .

4 **Look at the *Message* variable definition.**

The global variable calls the PFC's n_msg service.

5 **Close the *Application* property sheet.**
Select *File*>*Save* from the menu bar.

PowerBuilder saves the updated Application object.

6 **Select *File*>*Close* from the menu bar.**

The Application painter closes.

LESSON 2 Create the Frame Window

In a typical MDI application, you define a window whose type is MDI frame and open other windows as sheets within the frame. PFC provides w_frame, a frame window that includes many MDI features, including a status bar and a sheet manager.

In this lesson you will:

- Create a descendent frame window
- Define pre- and post-open processing
- Add script to open the frame window
- Run the application

How long will this lesson take?
About 15 minutes.

What will you learn about PFC?

- How to open a sheet using a string passed via the message router
- How to enable the status bar service
- How to enable the sheet management service
- How to connect to a database using functions provided by n_tr

Create a descendent frame window

Where you are
> Create a descendent frame window
Define pre- and post-open processing
Add script to open the frame window
Run the application

Now you will create a frame window by inheriting from w_frame. Then you will define a script for the pfc_Open script for the new frame window. PFC calls this event when the user selects File>Open from the menu bar. You can call it from other parts of the application as necessary to open sheet windows.

1 **Click the *Inherit* button in the PowerBar.**

The Inherit From Object dialog box displays.

2 **Select the *pfemain.pbl* file in the Libraries list box.**
Select *Windows* in the Objects of Type drop-down list.
Select *w_frame* in the Object list box and click *OK*.

The Window painter workspace displays.

3 **Type *PFC Tutorial Frame* in the Title box in the Properties view.**
Click the *Toolbar* tab in the Properties view.
Clear the *ToolbarVisible* check box.

You will not use a toolbar in the frame window, only in the sheet windows.

4 **Double-click *pfc_Open* in the Event List view.**

The Script view displays for the pfc_Open event. If a Script view is part of the stack with the Event List view, the Script pane will display in place of the Event List pane.

Finding the pfc_Open event in the Event List view
The events in the Event List view (and Script view) are alphabetized in two different series—events with code are listed before events without code. The pfc_Open event is not precoded in any of its ancestor scripts.

5 **Type these lines in the Script view:**

```
String   ls_sheet
w_sheet  lw_sheet

ls_sheet = Message.StringParm
OpenSheet(lw_sheet, ls_sheet, this, 0, Layered!)
```

This script will open an instance of the sheet window specified in the passed StringParm. You will initialize the StringParm value for two sheet windows later in this tutorial.

Define pre- and post-open processing

Where you are
Create a descendent frame window
> Define pre- and post-open processing
Add script to open the frame window
Run the application

Now you will define two scripts: one for the pfc_PreOpen event to enable the sheet manager and status bar services, and one for the pfc_PostOpen event to connect to the database.

1 **Select *pfc_PreOpen* from the second drop-down list in the Script view and type these lines:**

```
this.of_SetSheetManager(TRUE)
this.of_SetStatusBar(TRUE)
this.inv_statusbar.of_SetTimer(TRUE)
```

This script enables the sheet management service, which provides the ability to minimize all sheets and undo the last sheet arrange command. It also enables the status bar service, displaying date and time information in the bottom-right corner of the frame.

2 **Select *pfc_PostOpen* from the second drop-down list in the Script view and type these lines:**

```
Integer  li_return
String   ls_inifile

ls_inifile = gnv_app.of_GetAppIniFile()

IF SQLCA.of_Init(ls_inifile,"Database") = -1 THEN
   MessageBox("Database", &
   "Error initializing from " + ls_inifile)
 HALT CLOSE
END IF
IF SQLCA.of_Connect() = -1 THEN
 MessageBox("Database", &
   "Unable to connect using " + ls_inifile)
 HALT CLOSE
ELSE
 this.SetMicroHelp  ("Connection complete")
END IF
```

This script accesses database connection parameters with the transaction service (n_tr) of_Init function and connects with the transaction service of_Connect function. If these functions succeed, the script displays a message in the status bar.

Extending the ancestor script
The purple icon in front of the pfc_postopen event indicates that an ancestor window is scripted for this event. By default, events are extended: script is processed first from ancestor objects, then from descendent objects. You can examine this script by selecting pfc_w_frame from the third drop-down list in the Script view.

3 **Select *File*>*Save* from the menu bar.**

The Save Window dialog box displays.

4 **Make sure *my_pfc_app.pbl* is selected in the Application Libraries list box.**
Type *w_tut_frame* in the Windows box.
Type the following comment in the Comments box:

```
Frame window for the PFC tutorial application.
```

5 **Click *OK*.**
Select *File*>*Close* from the menu bar.

PowerBuilder saves the window and you close the Window painter.

Add script to open the frame window

Where you are
Create a descendent frame window
Define pre- and post-open processing
> Add script to open the frame window
Run the application

Now you will add code to the application manager pfc_Open event to open the frame window.

1 **Click the *Open* button in the PowerBar.**
Select *my_pfc_app.pbl* in the Libraries list box.
Select *User Objects* in the Objects of Type drop-down list.

The n_cst_tutappmanager user object is the only user object in this library. It is selected in the Object list box.

2 **Click *OK*.**

The User Object painter workspace displays.

3 **Make sure *n_cst_tutappmanager* displays in the first drop-down list in the Script view.**
Select *pfc_Open* from the second drop-down list.
Type these lines for the pfc_Open script:

```
this.of_splash (1)
Open(w_tut_frame)
```

The first line opens the PFC splash screen. The second line opens the MDI frame window for the application. The splash screen will stay open for one second after the n_cst_tutappmanager establishes the database connection.

4 **Select *File*>*Save* from the menu bar.**

PowerBuilder saves the script changes.

5 **Select *File*>*Close* from the menu bar.**

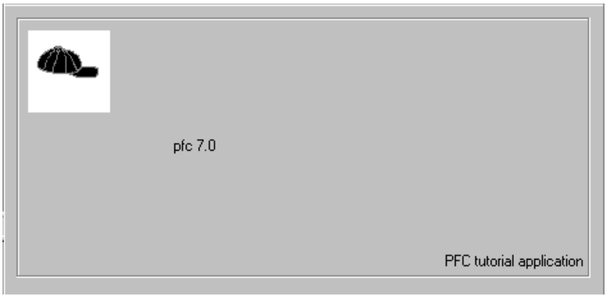
The User Object painter closes.

Run the application

Where you are
Create a descendent frame window
Define pre- and post-open processing
Add script to open the frame window
> Run the application

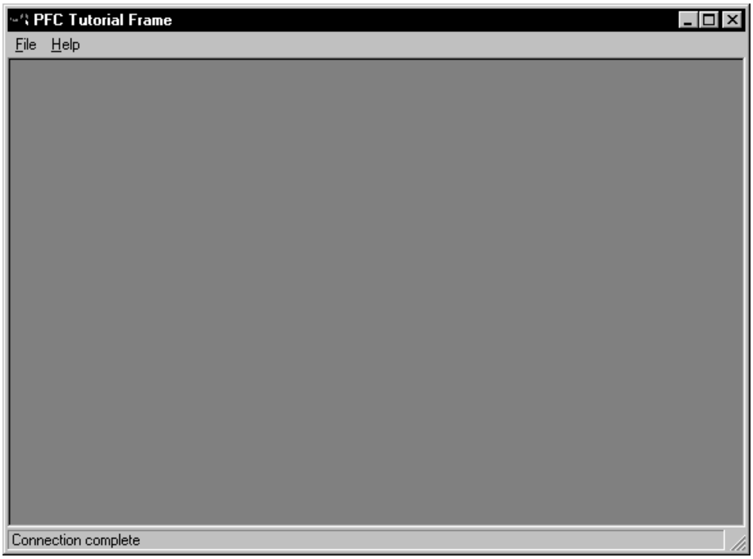
1 **Click the *Run* or the *Select And Run* button in the PowerBar. Make sure the *my_pfc_app* target is selected and click *OK*.**

The splash window displays and the database connection is established. The splash window uses information you entered for instance variables of the n_cst_tutappmanager user object:


pfc 7.0
PFC tutorial application

If there is a database connection error, you may need to modify the *pfctutor.ini* file to specify a valid data source for EAS Demo DB V3.

The MDI frame window displays behind the splash window and remains open after the splash window closes.



Connection complete

2 **Select** ***File*****>*****Exit*** **from the menu bar.**

The application closes.

LESSON 3

Create Menus

In PFC applications, all menus typically inherit from m_master (or an m_master descendant) and add, modify, and hide items as needed. The m_master ancestor menu provides items for use with all PFC windows, DataWindows, and visual controls.

Menu usage
This tutorial uses one approach to menu implementation. PFC allows you to implement other approaches, include modifying m_master directly and defining menus from scratch.

In this lesson you will:

- Create a descendent menu
- Add and modify items
- Create a frame menu
- Associate the frame window with a menu
- Create a menu for the w_products sheet
- Create a menu for the w_product_report sheet

How long will this lesson take?
About 40 minutes.

What will you learn about PFC?

- How to create a customized descendant of m_master
- How to open a sheet, passing the window name in Message.StringParm

Create a descendent menu

Where you are
> Create a descendent menu
Add and modify items
Create a frame menu
Associate the frame window with a menu
Create a menu for the w_products sheet
Create a menu for the w_product_report sheet

Now you will create a master menu for the tutorial by inheriting from the m_master menu.

1 **Click the *Inherit* button in the PowerBar.**

The Inherit From Object dialog box displays.

2 **Select *pfewnsrv.pbl* in the Libraries list box.**
Select *Menus* in the Objects of Type drop-down list.
Select *m_master* and click *OK*.

The Menu painter workspace displays.

3 **Select *File*>*Save As* from the menu bar.**

The Save Menu dialog box displays.

4 **Select *my_pfc_app.pbl* in the Application Libraries list box.**
Type *m_tut_master* in the Menus box.
Type the following comment in the Comments box:

```
Menu ancestor for tutorial frame and sheet menus.
```

5 **Click *OK*.**

PowerBuilder saves the menu.

Add and modify items

Where you are

Create a descendent menu
> Add and modify items
Create a frame menu
Associate the frame window with a menu
Create a menu for the w_products sheet
Create a menu for the w_product_report sheet

Now you will add and modify menu items on m_tut_master. You will use the Script, WYSIWYG, and Properties views of the Menu painter to make these changes.

Modifying m_master directly
The PFC tutorial creates an m_master descendant, which you modify for use as a master menu. In most applications you can make these modifications to m_master, eliminating a layer of inheritance.

1 **Select *m_file.m_open* in the first drop-down list in the Script view.**

This is the m_master (and m_tut_master) name for the File>Open menu item.

2 **Select the *Clicked* event in the second drop-down list in the Script view.**

The purple icon in front of the event name indicates that the Clicked event in an ancestor menu is scripted. By default, events are extended: script is processed first from ancestor objects, then from descendent objects.

3 **Select *Edit>Extend Ancestor Script* in the PowerBuilder menu bar.**

You clear the checkmark in front of the Extend Ancestor Script menu item. This will allow you to open a submenu with the application File>Open menu command without processing the ancestor script for the Clicked event of the Open menu item.

4 **Type this comment in the Script view:**

```
// File->Open script override.
```

5 **Click the File menu in the WYSIWYG view.**
Right-click the *Open* menu item below the File menu.
Select *Insert Submenu Item* from the pop-up menu.

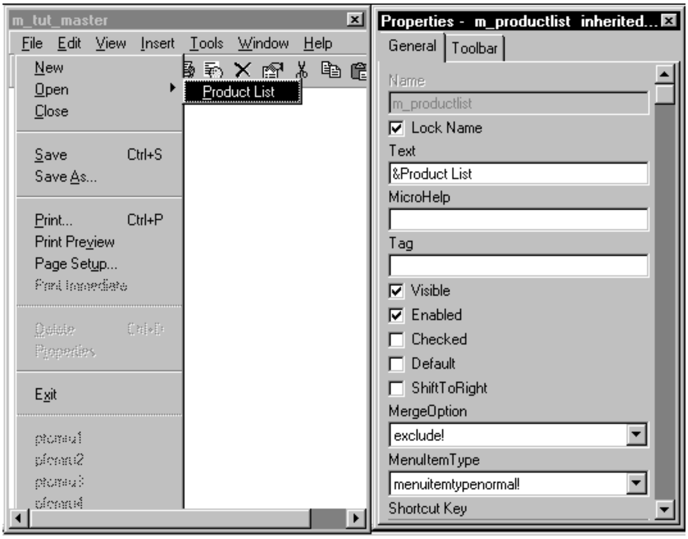
An empty box displays next to Open menu item. The cursor flashes inside the box, prompting you to define a submenu item for File>Open.

6 **Type *&Product List* in the empty box and press ENTER.**

The ampersand (&) converts the character that follows it to a menu hot key. After you press ENTER, the Product List menu item in the WYSIWYG view appears as it will at runtime: with an underscore under the P, and without the ampersand.

In the Text box in the Properties view, the ampersand is not replaced by the underscore. The Name box in the Properties view displays the normalized menu name: m_productlist.

Using the Properties view to enter the item text
If you click away from the WYSIWYG view before typing the menu item name, the box for the name will be blackened and you will not be able to modify it. In this case, clear the Lock Name box in the Properties view and type &Product List in the Text box in the Properties view. The value in the Name box will be changed automatically when you click elsewhere in the painter.



Name
m_productlist
☑ Lock Name
Text
&Product List
MicroHelp

Tag

☑ Visible
☑ Enabled
☐ Checked
☐ Default
☐ ShiftToRight
MergeOption
exclude!
MenuItemType
menuitemtypenormal!
Shortcut Key

7 **Type** *Product list* **for the MicroHelp box in the Properties view. Double-click** *Product List* **in the WYSIWYG view.**

The Script view displays m_file.m_open.m_productlist in the first drop-down list. When you created the Product List menu item, PowerBuilder added it to the list.

8 **Make sure the** *Clicked* **event is selected in the second drop-down list in the Script view.**
Type these lines:

```
Message.StringParm = "w_products"
of_SendMessage("pfc_Open")
```

These lines initialize the StringParm value and call the of_SendMessage menu function, which then calls the pfc_Open event on the frame window.

Menu service
You can also call the n_cst_menu of_SendMessage function to perform this functionality.

9 **Right-click the** *Product List* **menu item in the WYSIWYG view.**
Select *Insert Menu Item At End* **from the pop-up menu.**

An empty box displays below the Product List menu item.

10 **Type** *Product &Sales Report* **for the new menu item and click ENTER.**
Type *Product sales report* **in the MicroHelp in the Properties view.**
Double-click *Product Sales Report* **in the WYSIWYG view.**

The Script view displays m_file.m_open.m_productsalesreport in the first drop-down list.

11 **Make sure the Clicked event is selected in the second drop-down list.**
Type the following lines:

```
Message.StringParm = "w_product_report"
of_SendMessage("pfc_Open")
```

These lines initialize the StringParm value and call the of_SendMessage menu function, which then calls the pfc_Open event on the frame window.

12 **Click the** *File>New* **menu item in the WYSIWYG view.**

The Properties view displays properties of the m_new menu.

**13 Clear the *Visible* check box on the General page of the Properties view.
Select the *Toolbar* tab of the Properties view.
Clear the *ToolbarItemVisible* check box on the Toolbar page.**

Since you will not be adding code to the File>New menu Clicked event in the tutorial application, you make the menu item and its toolbar picture invisible at runtime.

In the previous lesson you made a selection to prevent the display of a toolbar for the application frame window. But the user can still opt to display a frame menu toolbar from an application that uses PFC window services. Now you ensure that even if the user makes this selection, the File>New toolbar picture will not be displayed.

Dithered appearance in WYSIWYG view

Menu items that will not be visible at runtime are displayed in a dithered format in the WYSIWYG view. Even though the menu item will not be visible, if its property sheet shows a toolbar picture for the menu item, you must clear the visible button on the toolbar page, or the toolbar picture will be displayed (or displayable by user selection) at runtime.

**14 Click the *Insert* menu bar item in the WYSIWYG view.
Clear the *Visible* check box on the General page of the Properties view.**

Menu items under the Insert menu will not be selectable at runtime, even though their Visible property is set to TRUE.

15 Hide menu items and toolbar pictures as follows:

Menu	Menu item	Hide item	Hide toolbar picture
File	New	Yes (done)	Yes (done)
	Open	No	Yes
	Save As	Yes	Not visible by default
Edit	Paste Special	Yes	No picture to hide
	m_dash23 (separator line below Select All)	Yes	No picture to hide
	Find	Yes	Yes
	Replace	Yes	Yes
	m_dash24 (separator line below Replace)	Yes	No picture to hide
	Update Links	Yes	No picture to hide
	Object	Yes	No picture to hide

Menu	Menu item	Hide item	Hide toolbar picture
View	Ruler	Yes	No picture to hide
	m_dash31 (separator line below Ruler)	Yes	No picture to hide
	Filter	Yes	No picture to hide

Hiding the items and toolbars separately
It may be quicker to go down the list once to hide the menu items and a second time to hide the toolbars. This way you avoid clicking back and forth between the General page and the Toolbar page of the Properties view.

16 Click the *Save* button in PainterBar1.
Click the *Close* button in PainterBar1.

PowerBuilder saves the updated menu.

Create a frame menu

Where you are
Create a descendent menu
Add and modify items
> Create a frame menu
Associate the frame window with a menu
Create a menu for the w_products sheet
Create a menu for the w_product_report sheet

Now you will create a frame menu by inheriting from the m_tut_master menu. The main portion of this exercise is hiding menus and menu items that don't apply when only the frame is displayed.

1 **Click the *Inherit* button in the PowerBar.**

The Inherit From Object dialog box displays.

2 **Make sure *my_pfc_app.pbl* is selected in the Libraries list box.**
Make sure *Menus* is selected in the Objects of Type drop-down list.
Select *m_tut_master* in the Object list box and click *OK*.

The Menu painter workspace displays.

3 **Click the *Save* button in PainterBar1.**

The Save Menu dialog box displays.

4 **Type *m_tut_frame* in the Menus box.**
Type the following comment in the Comments box and click *OK*:

```
Frame menu for PFC tutorial application.
```

PowerBuilder saves the menu and displays the Menu painter workspace.

5 **Using the WYSIWYG view and the Properties view, hide the following menu bar items: *Edit* menu, *View* menu, and *Tools* menu.**

The frame menu bar for the tutorial application will only show the File, Window, and Help menus.

6 **Hide menu items and toolbar button pictures as follows:**

Menu	Menu item	Hide item	Hide toolbar picture
File	Close	Yes	Not visible by default
	Save	Yes	Yes
	m_dash12 (separator line below Save As)	Yes	No picture to hide
	Print	Yes	No picture to hide
	Print Preview	Yes	Yes
	Print Immediate	Hidden in ancestor	Yes
Edit	Undo	Not necessary, Edit menu hidden	Yes
	Cut	Not necessary, Edit menu hidden	Yes
	Copy	Not necessary, Edit menu hidden	Yes
	Paste	Not necessary, Edit menu hidden	Yes
	Clear	Not necessary, Edit menu hidden	Not visible by default

7 **Select *File*>*Save* from the menu bar.**

PowerBuilder saves the updated menu.

8 **Select *File*>*Close* from the menu bar.**

The Menu painter closes.

Associate the frame window with a menu

Where you are
Create a descendent menu
Add and modify items
Create a frame menu
> Associate the frame window with a menu
Create a menu for the w_products sheet
Create a menu for the w_product_report sheet

Now you will associate the frame window with m_tut_frame, the frame menu you just created.

1 **Click the *Open* button in the PowerBar.**

The Open dialog box displays.

2 **Select *my_pfc_app.pbl* in the Libraries list box.**
Select *Windows* in the Objects of Type drop-down list.
Select *w_tut_frame* in the Object list box and click *OK*.

The Window painter workspace displays.

3 **Click the *ellipsis* button next to the MenuName box in the Properties view.**

The Select Object dialog box displays:

4 **Select *m_tut_frame* and click *OK*.**

The Window property sheet displays with the selected menu.

5 **Select *File*>*Save* from the menu bar.**

PowerBuilder saves the updated window.

6 **Select *File*>*Close* from the menu bar.**

The Window painter closes.

Create a menu for the w_products sheet

Where you are
Create a descendent menu
Add and modify items
Create a frame menu
Associate the frame window with a menu
> Create a menu for the w_products sheet
Create a menu for the w_product_report sheet

Now you will create a sheet menu by inheriting from m_tut_master, the PFC tutorial master menu.

1 **Click the *Inherit* button in the PowerBar.**

The Inherit From Object dialog box displays.

2 **Select *my_pfc_app.pbl* in the Libraries list box.**
Select *Menus* in the Objects of Type drop-down list.
Select *m_tut_master* and click *OK*.

The Menu painter workspace displays. You will leave the Edit, View, and Tools menus visible for the sheet menu.

3 **Hide menu items and toolbars as follows:**

Menu	Menu item	Hide item	Hide toolbar picture
File	Print	Yes	No picture to hide
	Print Preview	Yes	Yes
View	m_dash35 (separator line below Last)	Yes	No picture to hide
	m_dash36 (separator line below Filter)	Yes	No picture to hide
	Zoom	Yes	No picture to hide

4 **Select *File*>*Save As* from the menu bar.**

The Save Menu dialog box displays.

5 **Type** *m_products* **in the Menus box.**
Type the following line in the Comments box and click *OK***:**

```
Sheet menu for w_products window.
```

PowerBuilder saves your new menu as a descendant of m_tut_master.

Create a menu for the w_product_report sheet

Where you are
Create a descendent menu
Add and modify items
Create a frame menu
Associate the frame window with a menu
Create a menu for the w_products sheet
> Create a menu for the w_product_report sheet

Now you will create a sheet menu for w_product_report by inheriting from m_tut_master (the PFC tutorial master menu).

1 Click the *Inherit* button in the PowerBar.

The Inherit From Object dialog box displays.

2 Select *my_pfc_app.pbl* in the Libraries list box.
Select *Menus* in the Objects of Type drop-down list.
Select *m_tut_master* and click *OK*.

The Menu painter workspace displays.

3 Hide the *Edit* menu bar item.

You will prevent the user from modifying the product reports or updating the database at runtime.

4 Hide menu items and toolbar pictures as follows:

Menu	Menu item	Hide item	Hide toolbar picture
File	Save	Yes	Yes
	m_dash12 (separator line below Save As)	Yes	No picture to hide
Edit	Undo	Not necessary, Edit menu hidden	Yes
	Cut	Not necessary, Edit menu hidden	Yes
	Copy	Not necessary, Edit menu hidden	Yes
	Paste	Not necessary, Edit menu hidden	Yes

Menu	Menu item	Hide item	Hide toolbar picture
	Clear	Not necessary, Edit menu hidden	Not visible by default
View	Sort	Yes	No picture to hide
	m_dash36 (separator line below Filter)	Yes	No picture to hide

5 **Select *File*>*Save As* from the menu bar.**

The Save Menu dialog box displays.

6 **Type *m_product_report* in the Menus box.**
Type the following line in the Comments box and click *OK*.

```
Sheet menu for w_products_report window.
```

PowerBuilder saves your new menu as a descendant of m_tut_master.

7 **Close the Menu painter.**

LESSON 4 Build the First Sheet Window

You inherit from PFC's w_sheet window to create MDI sheets.

In this lesson you will:

- Add a library to the library list
- Create a descendent window
- Add a DataWindow control
- Enable DataWindow services
- Retrieve rows
- Run the application

How long will this lesson take?
About 30 minutes.

What will you learn about PFC?

- How to create a descendant of the w_sheet window
- How to enable the DataWindow property, row selection, and row management services
- How to use the u_dw DataWindow control
- How to add database access and update functionality to u_dw user events

Add a library to the library list

Where you are
> Add a library to the library list
Create a descendent window
Add a DataWindow control
Enable DataWindow services
Retrieve rows
Run the application

You will now add a library to the application target library list. The library you will add contains DataWindow objects created for this tutorial.

1 **Right-click the *my_pfc_app* target on the Workspace page of the System Tree.**
Select *Properties* from the pop-up menu.

The Library List page of the Properties dialog box displays all the libraries in the target search path.

2 **Click the *Browse* button.**
Navigate to the *PFC\Tutorial* directory.
Select *pfctutor.pbl* from the directory file list and click *Open*.

The *pfctutor.pbl* library appears at the bottom of the Library Search Path list box.

3 **Click *OK*.**

Create a descendent window

Where you are
Add a library to the library list
> Create a descendent window
Add a DataWindow control
Enable DataWindow services
Retrieve rows
Run the application

Now you will create a sheet window by inheriting from the w_sheet window.

1 **Click the *Inherit* button in the PowerBar.**

The Inherit From Object dialog box displays.

2 **Select *pfemain.pbl* in the Libraries list box.**
Select *Windows* in the Objects of Type drop-down list.
Select *w_sheet* in the Object list box and click *OK*.

The Window painter workspace displays.

3 **Type *Product List* in the Title box in the Properties view.**

This defines a title for the sheet window.

4 **Click the *ellipsis* button next to the MenuName box.**

The Select Object dialog box displays.

5 **Select *pfc_my_app.pbl* in the Application Libraries list box.**
Select *m_products* and click *OK*.

The Window painter redisplays with the MenuName box filled in.

6 **Select *File*>*Save As* from the menu bar.**

The Save Window dialog box displays. The pfc_my_app.pbl file is selected in the Application Libraries list box.

7 **Type *w_products* in the Windows box.**
Type the following line in the Comments box and click *OK*:

```
Sheet window for product list.
```

Add a DataWindow control

Where you are
Add a library to the library list
Create a descendent window
> Add a DataWindow control
Enable DataWindow services
Retrieve rows
Run the application

Now you will add a DataWindow control to the w_products window. This DataWindow control is based on PFC's u_dw DataWindow user object.

1 **Select and click the** *UserObj* **button in PainterBar1 (not the PowerBar).**
or
Select *Insert***>***Control***>***UserObject* **from the menu bar.**

If you don't see the UserObj button
To click the UserObj button, you first need to select the UserObj icon from a drop-down list of control buttons in PainterBar1 (this button typically displays a command button when the Window painter opens).

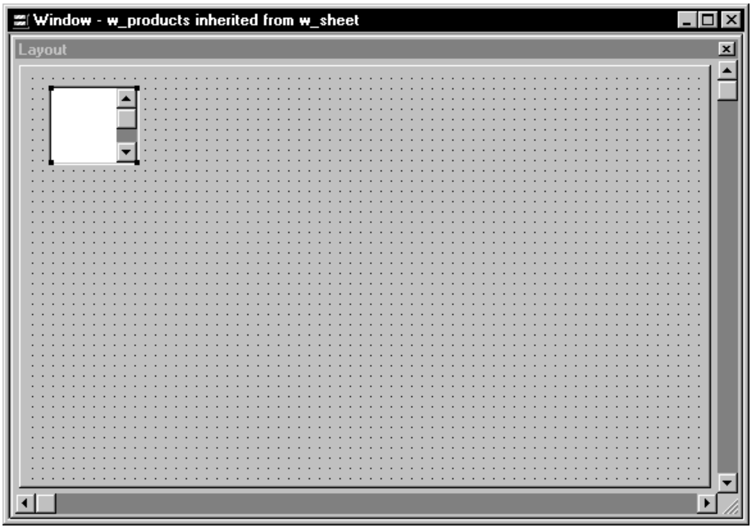
The Select Object dialog box displays.

2 **Select** *pfemain.pbl* **in the Application Libraries list box.**
Select *u_dw* **in the User Objects list box and click** *OK***.**

U_dw is a standard visual user object based on a DataWindow control that includes precoded events, instance variables, and functions to enable and disable PFC DataWindow services.

3 **Click in the upper-left corner of the window in the Layout view.**

PowerBuilder places a DataWindow control at the selected location. This DataWindow control is a descendant of u_dw, with access to u_dw events, functions, and instance variables:



4 **Select the text *dw_1* in the Name box of the Properties view. Type *dw_list* in the Name box.**

Make sure the new control is selected
If you do not see the name dw_1 in the Name box, click the control you just added in the Layout view. When the control is selected in the Layout view, it is also selected in the Properties view.

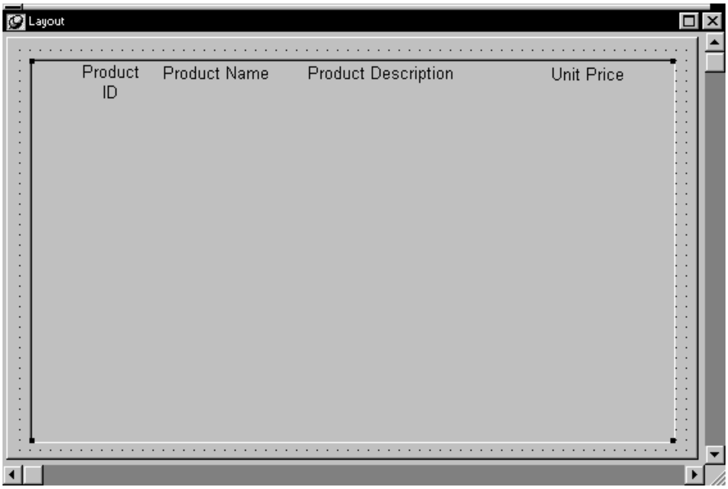
5 **Click the *ellipsis* button next to the DataObject box in the Properties view.**

The Select Object dialog box displays.

6 **Select *pfctutor.pbl* in the Application Libraries list box. Select *d_prodlist* in the DataWindows list box and click *OK*.**

The DataWindow painter redisplays.

7 **Make the control almost as big as the window in the Layout view, maximizing the Layout view if necessary:**



8 **Select** *File***>***Save* **from the menu bar.**

PowerBuilder saves the updated window.

Enable DataWindow services

Where you are
Add a library to the library list
Create a descendent window
Add a DataWindow control
> Enable DataWindow services
Retrieve rows
Run the application

Now you will use the Script view of the Window painter to add PowerScript code to the DataWindow control. The script you will add calls functions to enable PFC DataWindow sort, row selection, and row management services.

1 **Select *dw_list* from the first drop-down list in the Script view. Select the *Constructor* event from the second drop-down list.**

The third drop-down list in the Script view displays the parent window name, w_products. There is no code yet for the user object Constructor event, either in the current object or in the u_dw and pfc_u_dw ancestor objects.

2 **Type the following script for the Constructor event:**

```
this.of_SetRowSelect(TRUE)
this.of_SetRowManager(TRUE)
this.of_SetSort(TRUE)
this.of_SetProperty(TRUE)
this.of_SetTransObject(SQLCA)
```

These lines enable the property, row selection, row management, and sort services for the DataWindow. They also set the Transaction object for the DataWindow.

Using drag and drop from the System Tree
You can drag and drop methods and properties from the System Tree to the Script view. When you drag and drop a function such as of_SetRowSelect, PowerBuilder adds comments that serve as placeholders and give the data types for any arguments of the function.

3 **Add the following script after the lines you just typed:**

```
this.inv_rowselect.of_SetStyle &
  (dw_list.inv_rowselect.EXTENDED)
this.inv_sort.of_SetStyle  &
   (dw_list.inv_sort.DRAGDROP)
this.inv_sort.of_SetColumnHeader(TRUE)
```

These lines initialize the row selection and sort services.

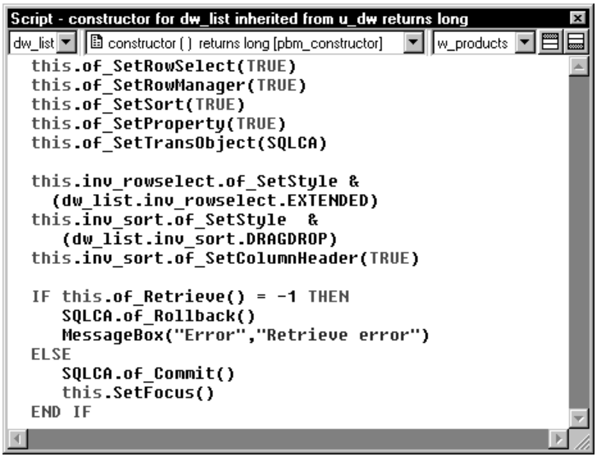
The row selection service of_SetStyle function enables extended row selection with the CTRL and SHIFT keys. The sort service of_SetStyle function instructs PFC to display a drag-and-drop sort dialog box when the user selects View>Sort from the menu bar.

The sort service of_SetColumnHeader function enables sorting by clicking on column headers, a feature found in many current applications.

4 **Add the following script after the lines you just typed:**

```
IF this.of_Retrieve() = -1 THEN
      SQLCA.of_Rollback()
      MessageBox("Error","Retrieve error")
ELSE
      SQLCA.of_Commit()
      this.SetFocus()
END IF
```

These lines call the of_Retrieve function for the user object. Since this function is not coded in the u_dw control, PowerBuilder will parse the code for the same function in the pfc_u_dw ancestor.


Script - constructor for dw_list inherited from u_dw returns long
dw_list
constructor ( ) returns long [pbm_constructor]
w_products

```
this.of_SetRowSelect(TRUE)
this.of_SetRowManager(TRUE)
this.of_SetSort(TRUE)
this.of_SetProperty(TRUE)
this.of_SetTransObject(SQLCA)

this.inv_rowselect.of_SetStyle &
  (dw_list.inv_rowselect.EXTENDED)
this.inv_sort.of_SetStyle  &
   (dw_list.inv_sort.DRAGDROP)
this.inv_sort.of_SetColumnHeader(TRUE)

IF this.of_Retrieve() = -1 THEN
   SQLCA.of_Rollback()
   MessageBox("Error","Retrieve error")
ELSE
   SQLCA.of_Commit()
   this.SetFocus()
END IF
```

5 **Click the *Compile* button in PainterBar2.**

PowerBuilder compiles the script you typed for the dw_list Constructor event.

Retrieve rows

Where you are
Add a library to the library list
Create a descendent window
Add a DataWindow control
Enable DataWindow services
> Retrieve rows
Run the application

Now you will add PowerScript code that retrieves rows from the database. In the last exercise, the call you made to the of_Retrieve function triggers the pfc_Retrieve event when the PFC Linkage service is not running. (You do not start the Linkage service in the tutorial application.)

Since events are extended by default, PowerBuilder parses the event script in both the pfc_u_dw ancestor, and then in the current u_dw control.

1 **Select *pfc_Retrieve* from the second drop-down list in the Script view.**

There is already script for this event in the pfc_u_dw ancestor control. Now you will add code to extend the ancestor script.

2 **Type this script:**

```
Return this.Retrieve( )
```

This line returns the Retrieve function return value, which gives the number of rows in the primary buffer if the retrieve is successful.

3 **Select *File>Save* from the menu bar.**

PowerBuilder compiles the script and saves the window.

4 **Select *File*>*Close* from the menu bar.**

The Window painter closes.

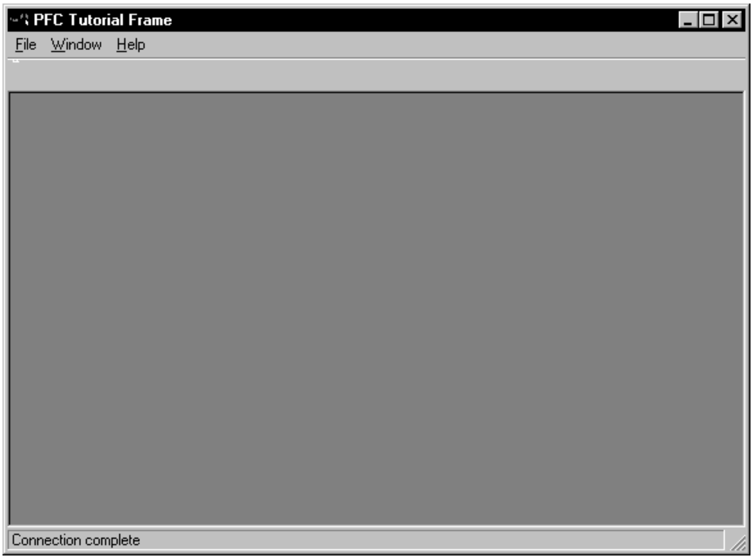
Run the application

Where you are
Add a library to the library list
Create a descendent window
Add a DataWindow control
Enable DataWindow services
Retrieve rows
> Run the application

Now you will make sure the sheet window opens properly by running the PFC tutorial application.

1 **Click the *Run* button in the PowerBar.**

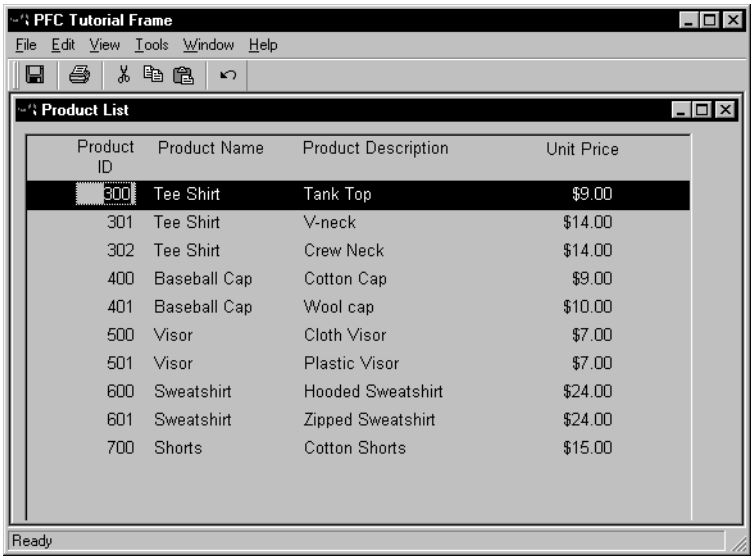
The application connects to the database and displays the PFC tutorial frame window. Only the File, Windows, and Help menus are visible in the MDI frame menu bar.



Connection complete

2 **Select *File*>*Open*>*Product List* from the menu bar.**

The w_products window displays:



Product ID	Product Name	Product Description	Unit Price
300	Tee Shirt	Tank Top	$9.00
301	Tee Shirt	V-neck	$14.00
302	Tee Shirt	Crew Neck	$14.00
400	Baseball Cap	Cotton Cap	$9.00
401	Baseball Cap	Wool cap	$10.00
500	Visor	Cloth Visor	$7.00
501	Visor	Plastic Visor	$7.00
600	Sweatshirt	Hooded Sweatshirt	$24.00
601	Sweatshirt	Zipped Sweatshirt	$24.00
700	Shorts	Cotton Shorts	$15.00

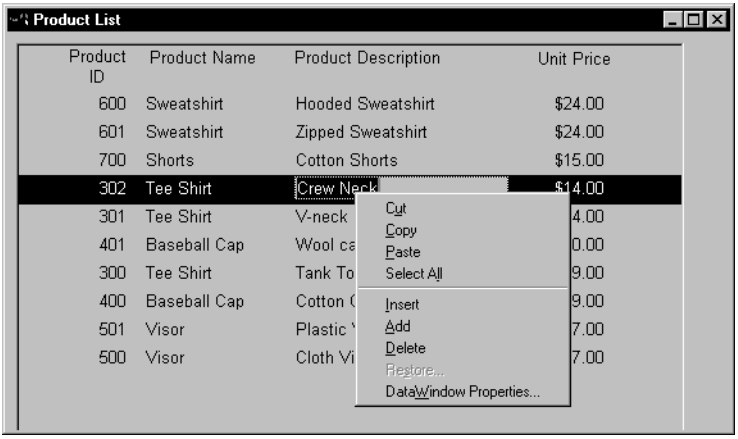
3 **Select multiple rows by holding the CTRL or SHIFT key down while clicking before the first column in the rows you want to select.** **Sort rows by clicking in the column headers.** **Click the same column header twice to change the order of the sort.**

You test the PFC selection and sort services.

Selecting multiple rows
You must click in front of the rows you want to select—clicking inside the rows does not select them.

4 **Click the right mouse button over one of the columns in the DataWindow.**

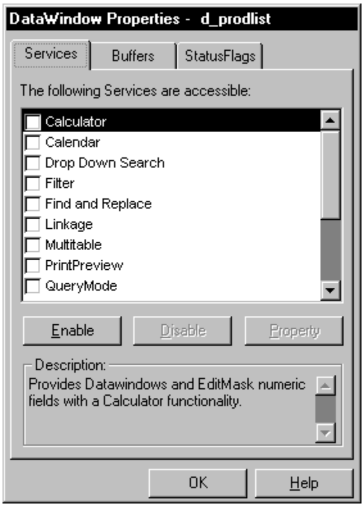
The DataWindow displays a pop-up menu, providing quick access to common actions:



Product ID	Product Name	Product Description	Unit Price
600	Sweatshirt	Hooded Sweatshirt	$24.00
601	Sweatshirt	Zipped Sweatshirt	$24.00
700	Shorts	Cotton Shorts	$15.00
302	Tee Shirt	Crew Neck	$14.00
301	Tee Shirt	V-neck	4.00
401	Baseball Cap	Wool ca	0.00
300	Tee Shirt	Tank To	9.00
400	Baseball Cap	Cotton (	9.00
501	Visor	Plastic '	7.00
500	Visor	Cloth Vi	7.00

5 Select *DataWindow Properties*.

The DataWindow Properties dialog box displays. You can enable or disable DataWindow services from this dialog box, and you can modify the properties of services that are enabled.



The following Services are accessible:

- □ Calculator
- □ Calendar
- □ Drop Down Search
- □ Filter
- □ Find and Replace
- □ Linkage
- □ Multitable
- □ PrintPreview
- □ QueryMode

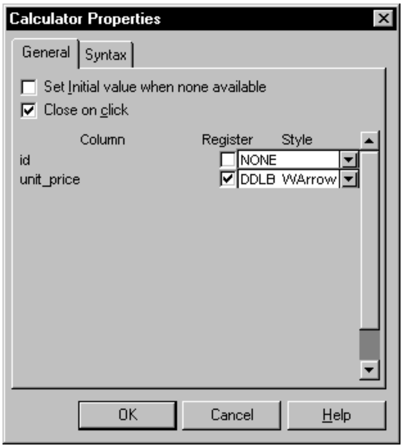
Description:
Provides Datawindows and EditMask numeric fields with a Calculator functionality.

6 Click the *Calculator* check box in the list of DataWindow services. Click the *Property* button.

If the Property button is grayed
The Property button is only enabled (not grayed) if the selected DataWindow service is enabled.

The Calculator Properties dialog box displays. The 2 columns that have numeric data types (id and unit_price) are listed on the General page of the Calculator Properties dialog box.

7 **Select the *Register* check box for the unit_price column.**
Select *DDLB With Arrow* from the drop-down list for the unit_price column.



□ Set Initial value when none available
☑ Close on click

Column	Register	Style
id	□	NONE
unit_price	☑	DDLB WArrow

8 **Click the *Syntax* tab.**

The Syntax page displays the PowerScript code that sets the properties you just selected.

9 **Click *OK* twice.**

You can now use a drop-down calculator in the Unit Price column to change column values.

Displaying the modified column values
The values entered with the drop-down calculator (or from the keyboard) will display with the appropriate column display formats—in this case, with a dollar sign and two decimal places—when the user clicks in a different row or column.

You can try enabling other DataWindow services and changing their properties.

10 **Select *File*>*Exit* from the menu bar.**

The runtime application closes.

LESSON 5

Build the Second Sheet Window

You inherit from PFC’s w_sheet window to create a second MDI sheet window.

In this lesson you will:

- Create a descendent window
- Add a DataWindow control
- Enable report and print preview services
- Run the application

How long will this lesson take?
About 10 minutes.

Create a descendent window

Where you are
> Create a descendent window
Add a DataWindow control
Enable report and print preview services
Run the application

Now you will create a sheet window by inheriting from the w_sheet window.

1 **Click the *Inherit* button in the PowerBar.**

The Inherit From Object dialog box displays.

2 **Click *pfemain.pbl* in the Libraries list box.**
Select *Windows* from the Objects of Type drop-down list.
Select *w_sheet* from the Object list box and click *OK*.

The Window painter workspace displays.

3 **Type *Product Sales Report* in the Title box in the Properties view.**

This defines a title for the new sheet window.

4 **Click the *ellipsis* button next to the MenuName box in the Properties view.**

The Select Object dialog box displays.

5 **Select *my_pfc_app.pbl* in the Application Libraries list box.**
Select *m_product_report* in the Menus list box and click *OK*.

The Window property sheet redisplays with the Menu Name field filled in.

6 **Select *File>Save As* from the menu bar.**

The Save Window dialog box displays.

7 **Type *w_product_report* in the Windows box.**
Type the following line in the Comments box and click *OK*.

```
This is the report sheet for the PFC tutorial.
```

Add a DataWindow control

Where you are
Create a descendent window
> Add a DataWindow control
Enable report and print preview services
Run the application

Now you will create a DataWindow control using PFC's u_dw DataWindow user object.

1 **Select and click the *UserObj* button in PainterBar1 (not the PowerBar).**
or
Select *Insert>Control>UserObject* from the menu bar.

The Select User Object dialog box displays.

2 **Select *pfemain.pbl* in the Application Libraries list box.**
Select *u_dw* in the User Objects list box and click *OK*.
Click in the upper-left corner of the window in the Layout view.

PowerBuilder places a DataWindow control at the selected location. This DataWindow control is a descendant of u_dw, with access to u_dw events, functions, and instance variables.

3 **Select the text *dw_1* in the Name box in the Properties view.**
Type *dw_report* in the Name box.
Select the *HScrollBar* check box.

You add a horizontal scroll bar to the DataWindow that will be visible at runtime.

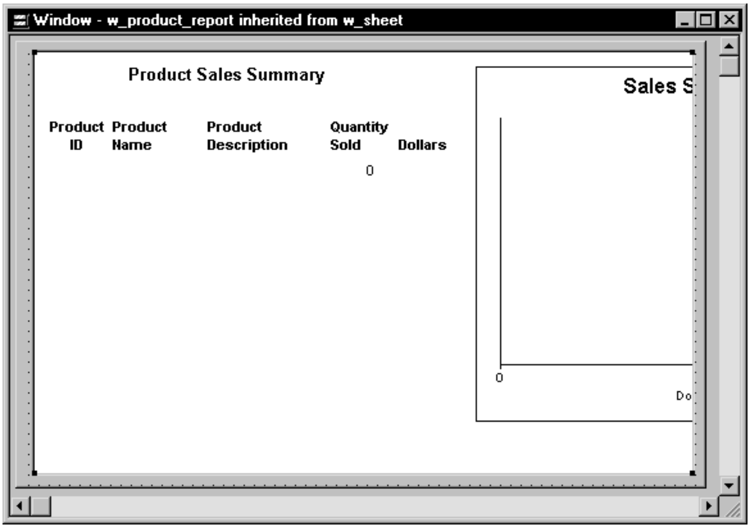
4 **Click the *ellipsis* button next to the DataObject box.**

The Select Object dialog box displays.

5 **Select *pfctutor.pbl* in the Application Libraries list box.**
Select *d_sales_report* in the DataWindows list box and click *OK*.

The DataWindow property sheet redisplays with the DataObject box filled in.

6 **Make the control almost as big as the window in the Layout view, maximizing the Layout view if necessary:**



7 **Select** *File>Save* **from the menu bar.**

Enable report and print preview services

Where you are
Create a descendent window
Add a DataWindow control
> Enable report and print preview services
Run the application

Now you will add PowerScript code to call functions that enable the PFC DataWindow report service and print preview service. You will also add code that retrieves rows from the database.

1 **Select *dw_report* from the first drop-down list in the Script view. Select the *Constructor* event from the second drop-down list. Type the following script for the Constructor event:**

```
this.of_SetReport(TRUE)
this.of_SetPrintPreview(TRUE)
this.of_SetTransObject(SQLCA)
this.of_SetUpdateable(FALSE)
```

These lines enable the report and print preview services, set SQLCA as the Transaction object and register the DataWindow as nonupdatable. In a nonupdatable DataWindow, PFC disregards default CloseQuery processing.

2 **Add the following script after the lines you just typed:**

```
IF this.of_Retrieve() = -1 THEN
      SQLCA.of_Rollback()
      MessageBox("Error","Retrieve error")
ELSE
      SQLCA.of_Commit( )
END IF
```

These lines call the u_dw of_Retrieve event for the DataWindow. Precoded report service events and functions handle all other processing.

3 **Select *pfc_Retrieve* from the second drop-down list in the Script view.**

PowerBuilder compiles the script for the Constructor event.

4 **Type this script for the pfc_Retrieve event:**

```
Return this.Retrieve( )
```

5 **Select** *File***>***Save* **from the menu bar.**

PowerBuilder compiles the script and saves the window.

6 **Select** *File***>***Close* **from the menu bar.**

The Window painter closes.

Run the application

Where you are
Create a descendent window
Add a DataWindow control
Enable report and print preview services
> Run the application

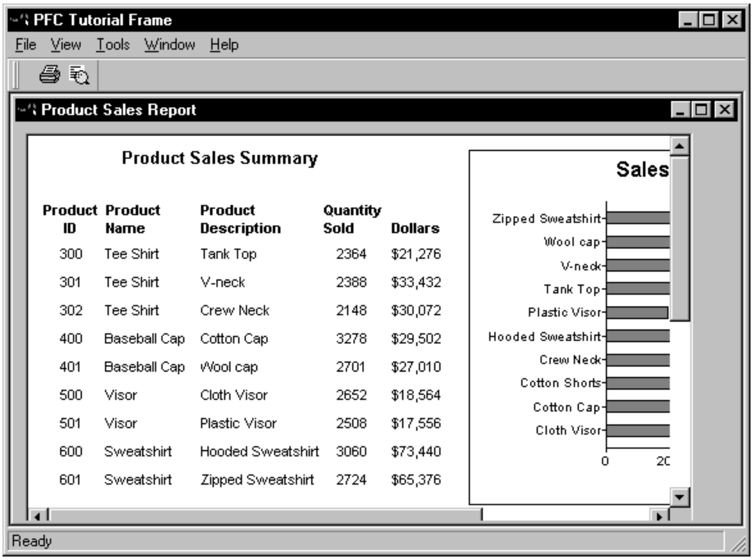
Now you will run the completed PFC tutorial application.

1 **Click the *Run* button in the PowerBar.**

The application connects to the database and displays the PFC Tutorial Frame window.

2 **Select *File*>*Open*>*Product Sales Report* from the menu bar.**

The w_product_report window displays:



Product Sales Summary

Product ID	Product Name	Product Description	Quantity Sold	Dollars
300	Tee Shirt	Tank Top	2364	$21,276
301	Tee Shirt	V-neck	2388	$33,432
302	Tee Shirt	Crew Neck	2148	$30,072
400	Baseball Cap	Cotton Cap	3278	$29,502
401	Baseball Cap	Wool cap	2701	$27,010
500	Visor	Cloth Visor	2652	$18,564
501	Visor	Plastic Visor	2508	$17,556
600	Sweatshirt	Hooded Sweatshirt	3060	$73,440
601	Sweatshirt	Zipped Sweatshirt	2724	$65,376

3 **Click the *Print Preview* button in the toolbar.**
or
Select *File>Print Preview* from the menu bar.

The print preview shows the printable area inside a blue box. You may need to scroll or zoom the preview window to see the entire report.

4 **Select *View>Zoom* from the menu bar.**

The Zoom dialog box displays.

If the Zoom dialog box does not display
You must be in the Print Preview mode to display the Zoom dialog box.

5 **Change the Zoom dialog box setting to display the entire report and click *OK*.**
Print the report if wanted.

6 **Select *File*>*Exit* from the menu bar.**

The application closes and the Window painter workspace displays.

7 **Close the Window painter.**

You have completed this tutorial. Before deploying a simple application like the one you created here, you would probably want to add your own Help file, edit the About box, and enable additional PFC services.

For more information on PFC functions and events, see the *PFC Object Reference*.

Using the Online Books
All the PowerBuilder books are available in the Technical Library CD and on the Sybase Web site at www.sybase.com.

Index